UNIVERSITÉ DU QUÉBEC À MONTRÉAL

JREGISTRE : UN REGISTRE UDDI EXTENSIBLE

MÉMOIRE

PRÉSENTÉ

COMME EXIGENCE PARTIELLE

DE LA MAÎTRISE EN INFORMATIQUE

PAR

RADHOUANE BEN TAMROUT

JANVIER 2006

Remerciements

Il m'est impossible de citer tous ceux qui ont contribué à la réalisation de ce travail. De ce fait, je commence par m'en excuser et par leur exprimer mes sincères remerciements.

Mes premiers et plus grands remerciements iront à Monsieur Hafedh Mili, professeur à l'Université du Québec à Montréal, pour m'avoir donné l'occasion de faire partie du laboratoire LATECE (LAboratoire de recherche sur les TEchnologies du Commerce Électronique), ainsi que pour avoir supervisé cette recherche. Sa grande disponibilité, sa persévérance, son souci du détail ainsi que ses encouragements m'ont beaucoup aidé tout au long de ce travail. Que ce mémoire lui soit le modeste témoignage de ma reconnaissance et de mon admiration.

Je remercie aussi tous mes collègues du groupe du LATECE, en particulier Ghizlane, Mohand, Abdeltif, Hamid, Wallès, Sonia, Guitta, Céline et Christian, qui ont supporté mes petites angoisses et mes remises en question. À tous mes amis, je tiens à vous remercier pour les bons moments que j'ai pu partager avec vous.

Je tiens aussi à exprimer ma profonde gratitude à toute ma famille, qui m'a toujours supporté dans les moments les plus difficiles et sans qui je n'aurais pas atteint ce niveau. Merci à vous tous.

Sameh, merci pour ta grande patience et pour ton soutien indéfectible dans tous les moments.

TABLE DES MATIÈRES

LISTE DES FIGURES

RÉSUMÉ

La mise en place de marchés électroniques, perçus comme des places publiques électroniques d'affaires où les partenaires ne se connaissant pas forcément, requiert la mise en place d'une sorte de registre public dans lequel les entreprises offrant un service donné viennent inscrire leurs services, et que les entreprises en quête de services viennent consulter. L'organisme de normalisation OASIS a proposé une norme pour la description et la fouille de services d'entreprises dans un registre public : la norme UDDI (Universal Description Discovery and Integration). Cependant, le type des requêtes supportées par cette norme reste primitif. Par exemple, la norme UDDI n'offre pas de mécanisme qui permette de choisir de manière intelligente un service Web en se basant sur des critères intrinsèques au service; on se base principalement sur des méta-données attribuées par un être humain. Plusieurs chercheurs ont proposé des extensions à UDDI pour supporter des requêtes de recherche plus complexes. Cependant, ces requêtes ne sont pas toujours exécutées au niveau du registre UDDI, et donc ne peuvent pas être partagées. Quand elles sont exécutées sur un registre UDDI, elles font défaut à la norme. Dans cette recherche, nous proposons une plate-forme d'extension générique de registre UDDI qui supporte, 1) l'ajout dynamique de nouvelles requêtes complexes, et 2) la co-existence de requêtes normalisées avec les requêtes étendues. Notre solution consiste en un intermédiaire (*broker*) qui agit comme un courtier entre les clients et les registres UDDI standards (Mili et al.,2005). Ce courtier peut être configuré à l'exécution pour supporter de nouvelles requêtes. Notre solution a comme avantages, 1) la rétrocompatibilité, et 2) l'extension dynamique. Nous décrivons notre implantation basée sur le registre jUDDI, une implantation de la fondation Apache du registre UDDI et AspectJ, une extension orientée aspect s de Java, développée par la fondation Eclipse™.

MOTS-CLES: Services Web, découverte et invocation dynamique, standard UDDI, extensions UDDI, jUDDI, AspectJ.

CHAPITRE I

Introduction

Les affaires électroniques, en tant que concept, existent depuis des décennies. Des entreprises liées fortement par une chaîne logistique stable collaborent depuis des décennies à l'aide d'échanges électroniques de documents. Des solutions comme EDI ont essayé, depuis plus d'un quart de siècle, de normaliser une façon d'échanger des informations dans un contexte d'affaires inter-entreprises. Le but principal était d'automatiser les échanges et les dialogues entre partenaires établies de longue date, ou en vue de relations d'affaires durables, justifiant l'investissement requis pour cette infrastructure d'échange. Cependant, la réalité d'aujourd'hui fait que les entreprises se doivent d'être capables d'intégrer de nouveaux partenaires à leurs systèmes informatiques de façon efficace, rapide et économique. Une entreprise veut, selon ses besoins, établir dynamiquement des liens avec divers partenaires afin d'utiliser les services que ces derniers offrent, avec un minimum de dépendance et d'investissement technologique.

L'Internet offre aux entreprises une infrastructure d'échange ouverte et peu dispendieuse, comparativement aux infrastructures dédiées requises jadis par l'EDI. De plus, l'avènement de normes d'échanges de documents (XML), et de distribution « légère » (services Web) ont ouvert la possibilité pour la mise en place de normes et outils leur facilitant les communications avec leurs partenaires et clients. L'une de ces normes, appelée UDDI, est conçue dans le but de permettre la découverte et l'invocation dynamique de services Web. Elle offre aux entreprises un mécanisme qui permet de découvrir des partenaires d'affaires dynamiquement. Cet objectif est encore loin de la réalité et ce genre d'interactions dynamiques est encore loin d'être réalisé. Plusieurs chercheurs ont proposé des extensions aux registres UDDI pour s'approcher de l'objectif. Seulement, ces extensions ne

sont pas forcément compatibles avec la norme, ni entre elles. Le but de notre travail est de proposer un cadre générique permettant d'étendre les fonctionnalités d'un registre UDDI existant de façon uniforme, cohérente avec la norme, et à l'exécution, càd sans arrêt du registre, ni effort de migration.

1.1. Découverte de partenaires d'affaires

Pour illustrer la problématique, prenons l'exemple concret d'un homme d'affaires qui veut organiser un voyage de Montréal à Tunis pour deux semaines. Il veut faire une réservation dans un hôtel quatre étoiles à Tunis pour les deux semaines et il veut que le prix soit raisonnable. Il demande à son agent de voyage de lui organiser ce voyage. L'agent effectue les opérations de recherche et de comparaison des prix et de réservation puis communique les résultats au client.

Pour pouvoir exécuter de telles requêtes, un système devrait, idéalement, être capable de réaliser les tâches suivantes :

- Découvrir les services Web qui permettent de consulter les tarifs et les disponibilités des billets d'avion aller-retour de Montréal à Tunis et qui font la réservation ;
- Découvrir les services Web qui permettent de consulter les tarifs et les disponibilités des chambres d'hôtel à quatre étoiles et qui font la réservation ;
- Sélectionner le service Web de réservation qui offre le meilleur tarif pour les billets d'avion aller-retour de Montréal à Tunis ;
- Sélectionner le service Web de réservation qui offre le meilleur tarif de chambre d'hôtel à quatre étoiles à Tunis ;
- Composer les services Web sélectionnés en un flux de processus permettant d'effectuer les réservations de billet d'avion et d'hôtel tout en spécifiant les recours en cas d'échec ;

- Exécuter le processus qui effectue les réservations ;

La découverte, la sélection, la composition ainsi que l'invocation des services Web doivent se faire de façon dynamique selon les besoins du client. Les interactions ne sont pas figées à l'avance, elles sont instanciées à la demande. Actuellement, on est encore loin de la concrétisation de ce scénario. Les techniques actuelles ne permettent pas à des services Web d'interagir de manière intelligente. Le processus de découverte qui représente l'élément clé de cette interaction est encore inefficace.

En effet, la découverte de services Web, assurée par les registres UDDI, est relativement primitive. Elle ne tient pas compte de la croissance continue du nombre de services Web offerts sur le Web et des modifications constantes subies par ces derniers. La norme UDDI permet de découvrir des services Web. Cependant, elle ne permet pas de choisir le meilleur fournisseur d'un service Web parmi plusieurs offrant le même service. Elle n'offre pas de mécanisme qui permette de choisir un service Web en se basant sur sa qualité. La norme manque aussi d'une sémantique compréhensible et interprétable par des machines capables de sélectionner le meilleur service Web disponible. De plus, UDDI n'offre pas de dispositif qui permette -ou même, qui aide- de composer dynamiquement un ensemble de services Web en une chaîne logistique. La définition des interactions et des règles d'affaires avec les partenaires se font encore de façon statique (Srivatava et Koehler., 2003). En somme, la description syntaxique proposée par la norme UDDI est insuffisante pour permettre l'automatisation du processus de découverte, de sélection et de composition de services Web.

1.2. Approches existantes

Plusieurs approches ont traité différents aspects liés aux registres UDDI. Ces approches partagent le but commun de rendre le processus de découverte, de composition et d'invocation de service Web plus flexible. Certaines de ces approches ont proposé d'ajouter une couche sémantique à UDDI (Paolucci et al., 2003) et (Colgrave et al., 2004). D'autres approches ont proposé d'intégrer des modèles de découverte de services Web basés sur la qualité de service (Qos) pour permettre de faire des recherches de services plus fines. L'inconvénient majeur de toutes ces approches réside dans le fait qu'elles nécessitent

l'implantation d'un nouveau registre UDDI. Par exemple, la solution apportée par Srinivasan et al. nécessite l'ajout d'un port d'entrée au registre UDDI pour permettre aux clients d'envoyer des requêtes de recherche de services Web basée sur une description sémantique de type OWL-S (Srinivasan et al., 2004).

D'autre part, certaines approches ont apporté de nombreux formalismes et techniques à UDDI essayant de faciliter la découverte ainsi que la composition dynamique de services Web en un processus d'affaires (Zhang et al., 2003 B) et (Srivatava et Koehler., 2003). Ces approches n'imposent pas de changer l'implantation des registres UDDI. Elles utilisent un intermédiaire qui fait un traitement local pour ensuite déléguer une partie de son travail à un ou plusieurs registre UDDI standard. Ce type d'approche est moins intrusif par rapport aux autres approches. Cependant, il donne naissance à un nouveau problème. Un client doit parler à différents APIs spécifiques à chacune des extensions qu'il veut utiliser (QoS, Composition dynamique, etc.). Il doit interagir avec plusieurs intermédiaires distants et doit gérer les interactions localement. L'utilisation de plusieurs types d'extension rend la tâche du client plus lourde.

1.3. Notre approche

Le but de notre recherche est de proposer un mécanisme d'extension générique qui permette d'ajouter plusieurs extensions à un registre UDDI sur demande. L'idée consiste en un registre intermédiaire qui supporte l'ajout de nouveaux types de requêtes non définis dans la version standard du registre UDDI. Chaque nouvel ajout représente une nouvelle extension au registre UDDI (Mili et al.,2005).

Sur le plan pratique, nous développons un registre baptisé sous le nom de « JRegistre » qui supporte l'ajout dynamique de requêtes spécifiques à l'exécution. Nous avons choisi de nous inspirer de la même structure et de la même logique de développement définie dans le registre jUDDI développé par la fondation Apache (jUDDI, 2003).

1.4. Plan du mémoire

Dans le *deuxième* chapitre, nous définirons les concepts de base d'un service Web. Par la suite, nous présenterons XML et les Schémas XML qui sont à la base des standards liés aux services Web. Enfin, nous présenterons les standards SOAP, WSDL et UDDI.

Dans le *troisième* chapitre, nous présenterons différentes initiatives qui ont traité différentes extensions liées aux registres UDDI. Nous ferons une étude des travaux existants en nous basant sur une classification par problème résolu.

Dans le *quatrième* chapitre, nous décrirons notre approche. Nous commencerons par une description globale du principe de notre approche. Par la suite, nous présenterons la structure et le mode de fonctionnement du registre jUDDI. Ensuite nous décrirons l'architecture détaillée de notre registre. Enfin, nous présenterons en détail le processus d'intégration de nouvelles requêtes à supporter par le registre.

Dans le *cinquième* chapitre, nous décrirons l'aspect dynamicité de notre registre, i.e. la possibilité du supporter de nouveaux types de requêtes sur un registre existant sans devoir l'arrêter. En particulier, nous présenterons comment, une nouvelle requête est intégrée dynamiquement à notre registre.

Nous terminerons ce mémoire par une conclusion générale où nous ferons une synthèse des travaux réalisés jusqu'à présent, et nous présenterons les suites qu'il nous semble opportun de donner à nos travaux.

CHAPITRE II

La famille des standards pour les services web

Un service Web est une application logicielle accessible via le Web. Cette application interagit avec d'autres applications en utilisant des protocoles de communication basés sur XML, indépendamment des plates-formes et des langages sur lesquels elles reposent (Figure 2.1). La technologie des services Web se présente comme un nouveau paradigme d'architecture logicielle conçu pour faciliter l'accès aux applications au sein d'une entreprise, mais aussi entre entreprises (W3C- Web Services Architecture, 2001).

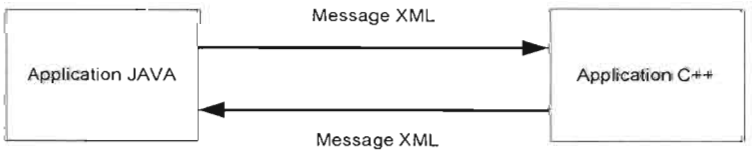

Figure 2.1. Échange de messages XML entre deux applications

Un ensemble de protocoles et standards reposants sur le langage de balisage XML sont apparus pour appuyer l'émergence des services Web. Le protocole SOAP définit la façon d'échanger des messages XML avec un service Web via Internet. Le standard WSDL fournit un mode de description des composants applicatifs d'un service Web. Les registres UDDI permettent d'exposer des services Web au grand public (Figure 2.2). Plusieurs autres standards font progressivement leur apparition et tentent de prendre en charge d'autres aspects liés aux services Web.

Le chapitre I sera axé sur la présentation des standards SOAP, WSDL et UDDI. D ans la section 1, seront présentés XML et les Schémas XML qui sont à la base des trois standards

sus-cités. Le standard SOAP est décrit dans la section 2. Suivra en section 3 une description de WSDL. La section 4 décrit, quant à elle, le standard UDDI.

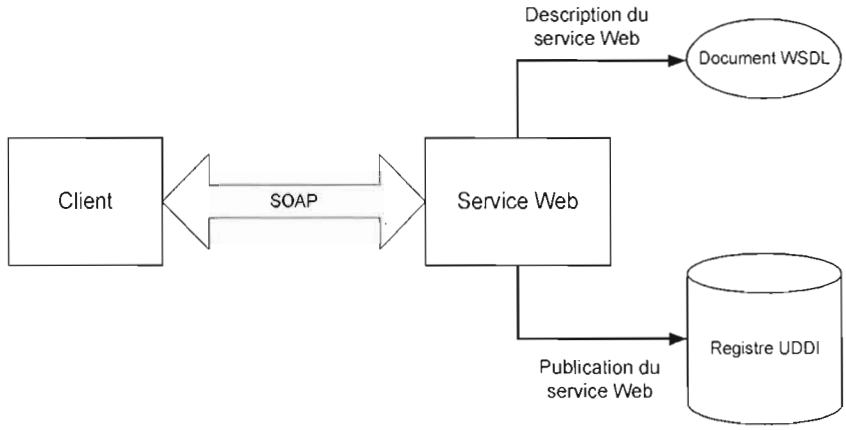

Figure 2.2. Protocoles et standards liés aux services Web

2.1. XML et les schémas XML

2.1.1. Le langage XML (eXtensible Markup Language)

XML, pour *eXtensible Markup Language* est un langage de balisage extensible qui définit un ensemble de règles permettant de structurer des données (Figure 2.3). Contrairement au langage HTML, XML est un langage ouvert. Il permet de créer ses propres balises selon le besoin. Ces balises n'ont pas de signification pour le langage XML, mais elles ont un sens pour les applications qui les utilisent. (W3C-XML, 2004)

```
<?xml version="1.0"?>
<additionner>
       <valeur1> 10 </valeur1>
       <valeur2> 5  </valeur2>
       <somme> 15  </somme>
</additionner>
```

Figure 2.3. Un simple document XML

Le standard XML permet de définir la structure d'un document sous forme de grammaire. Deux langages existent pour décrire ces grammaires : Les DTD (W3C-XML, 1998) et les schémas XML, connus sous le nom de XSD (W3C-XSD, 2000).

XML définit un mécanisme d'espaces de noms (W3C-nameSpaces, 2004) qui permet de créer des balises modulaires. Ce mécanisme donne aux éléments et aux attributs d'un document XML des noms uniques. Les espaces de noms peuvent être vus comme l'équivalent des packages en Java. En Java, il est possible de créer deux classes qui ont le même nom dans deux packages différents. C'est un dispositif qui donne à une classe un nom unique composé du nom de son package suivi de son nom. Deux classes ayant le même nom peuvent être utilisées dans une même application en faisant référence aux packages qui les contiennent. En XML, il est possible de définir le même nom de balise dans deux espaces de noms différents. Ces balises peuvent appartenir à différentes sources. Chacune de ces balises étant reconnue et associée à un contexte différent. Cela permet de créer des grammaires XML réutilisables.

Un espace de nom est identifié par l'attribut *xmlns* suivi par un *alias* qui servira comme préfixe aux balises et par un *URI* (*Uniform Resource Identifier*) qui fait référence au document d'origine définissant la structure de ces balises (Figure 2.4).

```
<?xml version="1.0"?>

<op:additionner xmlns:op="http//www.exemple.ca/operation.xsd">
  <op:valeur1> 10 </op:valeur1>
  <op:valeur2> 5  </op:valeur2>
  <op:somme> 15  </op:somme>
</op:additionner>
```

Figure 2.4. Les espaces de noms

Un espace de noms implicite (Figure 2.5) évite l'ajout du préfixe aux balises XML tout au long du document. Par défaut, tous les éléments fils d'un élément XML appartiendront au même espace de noms.

```
<?xml version="1.0"?>
<additionner xmlns="http//www.exemple.ca/operation.xsd">
  <valeur1> 10 </valeur1>
  <valeur2> 5  </valeur2>
  <somme> 15  </somme>
</additionner>
```

Figure 2.5. Les espaces de noms implicites

2.1.2. Les Schémas XML

Un schéma XML est un standard qui décrit la structure d'un document XML. Il définit les éléments et les attributs qui peuvent apparaître dans un document XML ainsi que l'ordre dans lesquels ils doivent apparaître (Figure 2.6). Contrairement à un DTD, un schéma XML est facile à lire. Il utilise le langage XML pour définir des contraintes de typage et les relations entre les éléments d'un document XML. (W3C-XSD, 2000)

```
<?xml version="1.0"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
           targetNamespace="http//www.exemple.ca/operation.xsd"
           xmlns="http//www.exemple.ca/operation.xsd">

  <xs:element name="additioner">
    <xs:complexType>
      <xs:sequence>
          <xs:element name="valeur1" type="xs:int"/>
          <xs:element name="from" type="xs:int"/>
      </xs:sequence>
    </xs:complexType>
  </xs:element>

</xs:schema>
```

Figure 2.6. Exemple d'un Schéma XML

Plusieurs technologies liées aux services Web utilisent les schémas XML pour définir leur structure. SOAP, WSDL et UDDI suivent chacun d'entre eux, un schéma XML défini par les créateurs de la technologie.

2.2. SOAP (Simple Object Acces Protocol)

SOAP, pour *Simple Object Acces Protocol* est un protocole de communication qui permet l'invocation de méthodes, de composants ou d'objets à distance par envoi de messages XML (W3C-SOAP, 2001). Ce protocole a été proposé à W3C en 2000 par un ensemble d'acteurs qui œuvrent dans le domaine informatique, dont IBM, Microsoft, SAP et bien d'autres. SOAP permet un échange inter-applications indépendamment de la plate-forme ou du langage d'implémentation utilisé. Il est utilisé au-dessus des protocoles de transport Internet comme HTTP, SMTP ou FTP. Actuellement, SOAP est à sa deuxième version. Dans ce qui suit, nous allons d'abord décrire la structure d'un message SOAP. Ensuite, nous allons présenter un exemple d'échanges de messages SOAP et les patrons d'échanges possibles. Enfin, nous présenterons de quelle manière est transporté un message SOAP par HTTP.

2.2.1. Structure d'un message SOAP

Un message SOAP est un document XML qui suit une structure définie dans un schéma XML. Il est composé principalement d'une enveloppe **(Envelope)** qui contient un en-tête **(Header)** optionnel et un corps **(Body)** obligatoire (Figure 2.7).

```
<?xml version="1.0"?>
<SOAP:Envelope xmlns:SOAP="URI du Schéma XML du protocole SOAP"
 SOAP:encodingStyle= "URI du style d'encodage du protocole SOAP" >
      <SOAP:Header>
                  En-tête optionnel
      </SOAP:Header>
      <SOAP:Body>
            Une requête SOAP
      </SOAP:Body>
</SOAP:Envelope>
```

Figure 2.7. Structure d'une requête SOAP

L'élément **Envelope**, racine d'un message SOAP, contient un attribut obligatoire **xmlns** qui fait référence au schéma XML décrivant la structure d'un message SOAP. Il contient aussi un attribut optionnel **encodingStyle** qui définit le style d'encodage du message SOAP.

L'élément **Header** permet de combiner d'autres technologies au protocole SOAP, telles que l'ajout des entrées pour le routage des messages SOAP, des informations relatives à

la sécurité, ainsi que d'autres types d'extensions. L'élément **Body** contient la requête SOAP sous format XML. Les messages SOAP sont généralement utilisés pour acheminer des requêtes SOAP d'une application à une autre, et les réponses le cas échéant. En général, un message SOAP qui contient une réponse (Figure 2.8) n'a pas de **header**.

```
<?xml version="1.0"?>
<SOAP:Envelope xmlns:SOAP="URI-Schema SOAP"
 SOAP:encodingStyle= " URI-Encoding-style SOAP " >
                    <SOAP:Body>
            Une réponse SOAP
      </SOAP:Body>
</SOAP:Envelope>
```

Figure 2.8. Structure d'une réponse SOAP

Le protocole SOAP gère aussi les exceptions. Dans le cas où un message SOAP contenant une requête qui ne peut être traitée par son receveur, un message d'erreur sous format XML (Figure 2.9) expliquant la nature du problème est retourné en réponse.

```
<?xml version="1.0"?>
<SOAP:Envelope xmlns:SOAP="URI-Schema SOAP"
 SOAP:encodingStyle= " URI-Encoding-style SOAP " >
      <SOAP:Body>
            <SOAP:Fault>
                  Détail de l'erreur (code erreur, cause …)
            <SOAP:Fault>
      </SOAP:Body>
</SOAP:Envelope>
```

Figure 2.9. Structure d'une réponse SOAP qui contient une erreur

2.2.2. Exemple d'échanges de messages SOAP

Pour illustrer un scénario d'échanges de messages SOAP, voici un exemple de service Web écrit en Java. Le service Web *Calculatrice* possède l'opération *additionner* qui permet l'addition de deux entiers (Figure 2.10).

```
public class Calculatrice {
            public int additionner (int valeur1, int valeur2)
                  {int somme = valeur1+valeur2
                   return somme;
                  }
}
```

Figure 2.10. Exemple Java d'un service Web

Pour pouvoir invoquer une opération d'un service Web, il faut au moins connaître le nom de l'opération à invoquer ainsi que le nom et la valeur de chacun de ses paramètres. Le message SOAP qui invoque l'opération *additionner* du service Web *Calculatrice* peut ressembler au message suivant :

```
<?xml version="1.0"?>
<SOAP:Envelope xmlns:SOAP="http://schemas.xmlsoap.org/soap/envelope"
      SOAP:encodingStyle="http://schemas.xmlsoap.org/soap/encoding">
  <SOAP:Body>
     <calc:additionner xmlns:calc="http//exemple.ca/Calculatrice" >
       <calc:valeur1> 10 </calc:valeur1>
       <calc:valeur2> 5  </calc:valeur2>
     </calc:additionner>
  <SOAP:Body>
</SOAP:Envelope>
```

Figure 2.11. Exemple d'une requête SOAP

Dans notre cas, l'élément **additionner** (Figure 2.11) représente le nom de la méthode à invoquer. Les sous éléments **valeur1** et **valeur2** représentent les noms des paramètres de cette méthode. La requête est envoyée au service Web et une réponse à l'invocation est reçue. On convient d'utiliser le nom « <requête>Response » pour représenter la réponse à une requête SOAP. D'autres conventions peuvent être utilisées.

```
<?xml version="1.0"?>
<SOAP:Envelope xmlns:SOAP="http://schemas.xmlsoap.org/soap/envelope"
      SOAP:encodingStyle="http://schemas.xmlsoap.org/soap/encoding">
   <SOAP:Body>
       <calc:additionnerResponse
             xmlns:calc="http//exemple.ca/Calculatrice">
       <calc:somme> 15 </calc:somme>
   </calc:additionner>
  <SOAP:Body>
</SOAP:Envelope>
```

Figure 2.12. Exemple d'une réponse SOAP

Il est possible d'envoyer une requête SOAP et ne pas recevoir une réponse. La section suivante décrit tous les scénarios d'échange possibles.

2.2.3. Patrons d'échange de messages SOAP

Le protocole SOAP supporte plusieurs patrons d'échange de messages SOAP:

- *Message de type requête-Réponse* : Le client envoie une requête SOAP au serveur et attend une réponse. Il est possible que le client reçoive plusieurs réponses pour une seule requête.
- *Message à sens unique (One-way)* : Le client envoie une requête SOAP au serveur et n'attend pas de réponse.
- *Message de type notification* : Le serveur envoie un message SOAP au client pour notification.
- *Message sollicité (serveur/client)* : Le serveur envoie un message SOAP au client et reçoit une réponse de ce dernier.

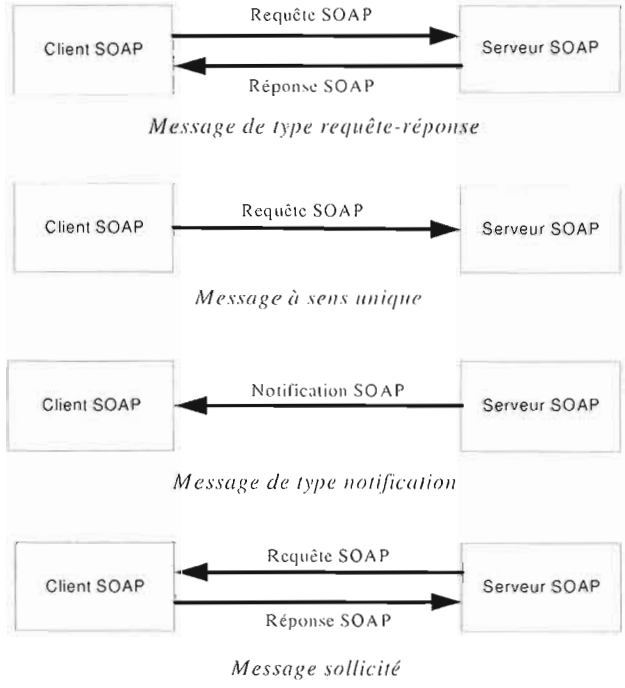

Figure 2.13. Patrons d'échange de messages SOAP

2.2.4. HTTP : Le transporteur idéal de SOAP

Le protocole SOAP ne précise pas le protocole de transport qu'il faut utiliser, mais la popularité de HTTP le rend la cible numéro un. En transportant SOAP sur HTTP, les administrateurs réseau n'auront pas à reconfigurer leur pare-feu puisque les ports 80 et 8080 utilisés par le protocole HTTP sont la plupart du temps ouverts sur les pare-feu.

Selon le document de spécification de la version 1.1 du protocole SOAP, l'échange HTTP est basé sur des requêtes de type POST (Figure 2.14). L'en-tête HTTP POST est composée d'une requête *Request-URI* qui identifie la destination de la requête HTTP sur le serveur, d'une adresse *Host* du serveur avec lequel on veut communiquer, d'un *Content-Type* qui indique que le corps de la requête est une présentation XML, d'un *SOAPAction* qui indique la destination de la requête SOAP et d'un corps de requête qui contient le message SOAP.

```
POST /Calculatrice HTTP/1.1
Host:exemple.ca
Content-Type:text/xml; charset="utf-8"
Content-Length:nnnn
SOAPAction:"http://exemple.ca/Calculatrice"
<?xml version='1.0' ?>
<SOAP:Envelope xmlns:SOAP=" " >
      Requête SOAP
</SOAP-ENV:Envelope>
```

Figure 2.14. Requête SOAP/HTTP

Une réponse (Figure 2.15) à une requête HTTP / SOAP indique si celle-ci a été traitée correctement ou non.

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
<?xml version='1.0' ?>
<SOAP:Envelope xmlns:SOAP=" " >
      Réponse  SOAP
</SOAP-ENV:Envelope>
```

Figure 2.15. Réponse SOAP/HTTP

Le protocole SOAP est considéré comme le standard de facto du monde des services Web. Les avantages qu'il fournit en terme de simplicité et d'interopérabilité sont indiscutables. Cependant, les experts de la sécurité n'hésitent pas à le qualifier de "monumental trou de sécurité". Ils considèrent que c'est assez facile de décoder et éventuellement de modifier les messages SOAP. Plusieurs intervenants ont essayé de résoudre ce problème. L'extension WS-Security (IBM-WS-Security, 2002) permet de mettre en place des informations de sécurité de base dans les headers SOAP.

Le protocole SOAP décrit une façon simple pour échanger des données avec un service Web. Il ne décrit ni les opérations offertes pas un service Web ni la structure des données à échanger. Dans la section suivante, nous introduisons le standard WSDL qui résout ce problème.

2.3. WSDL (Web Services Description Language)

Proposé initialement par IBM et Microsoft en 2001, WSDL est un langage de description de service Web sous format XML (W3C-WSDL, 2002). Il définit de façon abstraite et indépendante du langage de développement, i) l'ensemble des opérations et des messages qui peuvent être transmis vers et depuis un service Web donné, ii) les protocoles de communication et de transport utilisés et iii) les points d'accès au service. Un document WSDL décrit de façon détaillée comment utiliser un service Web (Figure 2.16). Actuellement WSDL est à sa version 2.0.

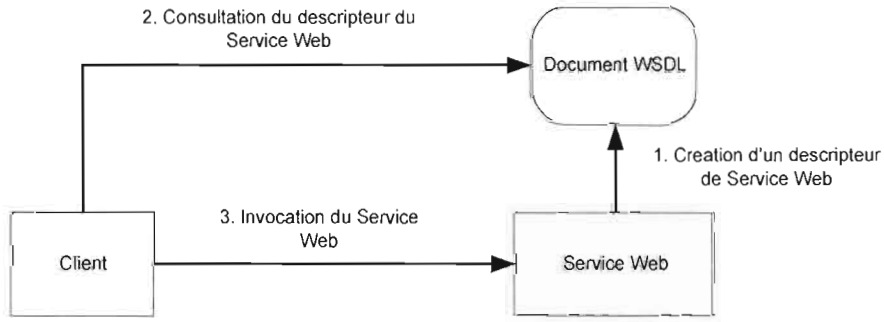

Figure 2.16. Consultation d'un document WSDL

2.3.1. Structure d'un document WSDL

Un document WSDL est un fichier XML, qui se conforme à un schéma XML. L'élément **`definitions`**, racine de ce document, englobe un ensemble d'éléments regroupés principalement en trois parties :

- Les éléments **`types`**, **`message`**, **`portType`** et **`operation`** définissent les opérations offertes par un service Web et les entrées-sorties de chacune de ces opérations.

- L'élément **`binding`** définit les protocoles de communication et de transport Internet utilisés pour invoquer les opérations définies dans l'élément **`portType`**.

- Les éléments **`service`** et **`port`** définissent les points d'accès au service.

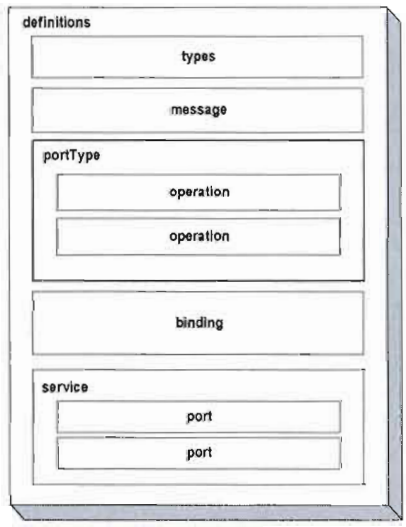

Figure 2.17. Structure globale d'un document WSDL

2.3.2. Description détailléé d'un document WSDL

Pour décrire en détail la structure XML d'un document WSDL, on va prendre en exemple le document WSDL (Figure 2.18) qui décrit le service Web *calculatrice* (Figure 2.10).

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="CalculatriceService"
             targetNamespace="http://exemple.ca/calculatrice.wsdl"
             xmlns:tns="http://exemple.ca/calculatrice.wsdl"
             xmlns="http://schemas.xmlsoap.org/wsdl/"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns:xsd="http://example.com/calculatrice.xsd" >

  <types>
       <schema targetNamespace="http://example.com/calculatrice.xsd"
               xmlns:xsd="http://www.w3.org/2000/10/XMLSchema">
           <xsd:element name="In" type="xsd:int">
           <xsd:element name="Out" type="xsd:int">
       </schema>
  </types>
  <message name="messageAdditioner">
      <part name="valeur1" element="xsd:In"/>
      <part name="valeur2" element="xsd:In"/>
  </message>
  <message name="messageAdditionerResponse">
      <part name="somme" element="xsd:Out"/>
  </message>
  <portType name="CalculatricePortType">
     <operation name="additioner" parameterOrder="valeur1 valeur2">
           <input message="tns:messageAdditioner"/>
           <output message="tns:messageAdditionerResponse"/>
     </operation>
  </portType>
  <binding name="CalculatriceBind" type="tns:CalculatricePortType">
    <operation name="additioner">
      <soap:operation soapAction="http://exemple.ca/calculatrice "/>
      <input>
           <soap:body use="encoded"/>
      </input>
      <output>
           <soap:body use="encoded"/>
      </output>
    </operation>
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
                  style="rpc"/>
  </binding>
  <service name="MyCalculatriceService">
    <port name="CalculatricePort" binding="tns:CalculatriceBind">
      <soap:address location=" http://exemple.ca/calculatrice "/>
    </port>
  </service>
</definitions>
```

Figure 2.18. Exemple d'un document WSDL

L'élément **definitions** (Figure 2.19) définit tous les espaces de noms utilisés dans le document WSDL.

```
<definitions name="Calculatrice"
             targetNamespace="http://exemple.ca/calculatrice.wsdl"
             xmlns:tns="http://exemple.ca/calculatrice.wsdl"
             xmlns="http://schemas.xmlsoap.org/wsdl/"
             xmlns:xsd="http://www.w3.org/2001/XMLSchema"
             xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
             xmlns:xsd="http://example.com/calculatrice.xsd" >
...
...

</definitions>
```

Figure 2.19. Définitions des espaces de nom

L'élément **types** (Figure 2.20) définit tous les types de données qui peuvent être échangés avec un service Web. La spécification WSDL n'exige pas un système de typage particulier, mais elle utilise par défaut les schémas XML.

```
<types>
       <schema targetNamespace="http://example.ca/calculatrice.xsd"
               xmlns:xsd="http://www.w3.org/2000/10/XMLSchema">
           <xsd:element name="In" type="xsd:int">
           <xsd:element name="Out" type="xsd:int">
       </schema>
</types>
```

Figure 2.20. Types de données

L'élément **message** (Figure 2.21) décrit un message unique qui peut être soit un message d'entrée soit un message de sortie d'un service Web. Chaque **message** est identifié par un nom et contient une ou plusieurs parties. Chaque partie est définie par l'élément **part**.

```
<message name="additionerMessage">
      <part name="valeurl" type="xsd:int"/>
      <part name="valeur2" type="xsd:int"/>
</message>
```

Figure 2.21. Structure d'un message

L'élément **portType** (Figure 2.22) définit de façon abstraite un ensemble d'opérations offertes par le service Web. Un document WSDL peut contenir plus d'un élément **portType**. Chaque **portType** est identifié par un nom et contient au moins une opération. Chaque élément **operation** est identifié par un nom et contient au plus un message d'entrée et au plus un message de sortie. Le même message peut exister dans plusieurs opérations. La nature de ces messages est définie selon le type de l'opération. L'élément **input** définit un message d'entrée pour l'opération, alors que sa sortie est définie par l'élément **output**. L'élément **fault** n'est pas présent dans l'exemple ci-dessous, il définit de la même façon qu'un élément **output**, un message d'erreur retourné par une opération en cas d'exception.

```
<portType name="CalculatricePortType">
      <operation name="additioner" parameterOrder="valeurl valeur2">
            <input message="tns:messageAdditioner"/>
            <output message="tns:messageAdditionerResponse"/>
      </operation>
</portType>
```

Figure 2.22. Structure d'un portType

La spécification WSDL décrit quatre types d'opérations :

- *Opération de type requête – réponse* : elle reçoit un **input** et envoie un **output**.
- *Opération à sens unique* : elle reçoit un **input** mais n'envoie aucun **output**.
- *Opération de type réponse sollicitée* : elle envoie un **output** et reçoit un **input**.
- *Opération de type notification* : elle envoie un **output** et ne reçoit pas d'**input**.

L'élément **binding** (Figure 2.23) définit les protocoles de communication et de transport Internet qui doivent être utilisés pour invoquer les opérations définies dans l'élément **portType**. Un document WSDL peut contenir plusieurs éléments **binding**. Chaque **binding** est identifié par un nom et fait référence à un **portType**. La spécification

WSDL ne précise pas les protocoles de communication et de transport qu'il faut utiliser; elle laisse le choix libre. Cependant, elle recommande l'extension SOAP qu'elle a définie afin d'utiliser SOAP comme protocole de communication et HTTP comme protocole de transport Internet. Dans ce cas, on décrit le style des messages SOAP reçus et envoyés par les opérations définies dans le **portType**. On spécifie, entre autre, s'il s'agit d'une invocation RPC ou d'un document XML. La construction des messages SOAP diffère d'un style à un autre.

```
<binding name="CalculatriceBind" type="tns:CalculatricePortType">
   <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
                 style="rpc"/>
   <operation name="additioner">
     <soap:operation soapAction="http://exemple.ca/calculatrice"/>
     <input>
           <soap:body use="encoded"/>
     </input>
     <output>
           <soap:body use="encoded"/>
     </output>
   </operation>
</binding>
```

Figure 2.23. Structure du binding

L'élément **service** (Figure 2.24) définit les points d'accès au service. Un élément **service** est identifié par un nom et contient au moins un élément **port**. Chaque élément **port** est identifié par un nom et définit un point d'accès au service Web.

```
<service name="MyCalculatriceService">
    <port name="CalculatricePort" binding="tns:CalculatriceBind">
        <soap:address location="http://exemple.ca/calculatrice"/>
    </port>
</service>
```

Figure 2.24. Structure du service

La spécification WSDL offre un mécanisme d'importation de documents qui permet de séparer les différents éléments d'un document WSDL en plusieurs documents indépendants.

L'élément **import** offre la possibilité d'importer des éléments définis ailleurs. Cela permet d'écrire des définitions plus claires et de maximiser la réutilisation des éléments définis.

Généralement, un document WSDL est séparé en deux parties (Figure 2.25). Une partie abstraite qui définit les opérations offertes par le service **(portType)** et les protocoles de communication et de transport utilisés **(binding)** ; et une partie concrète qui définit les points d'accès au service **(service)**. Il est possible d'ajouter des extensions n'importe où dans un document WSDL pour décrire d'autres aspects que la version native ne permet pas.

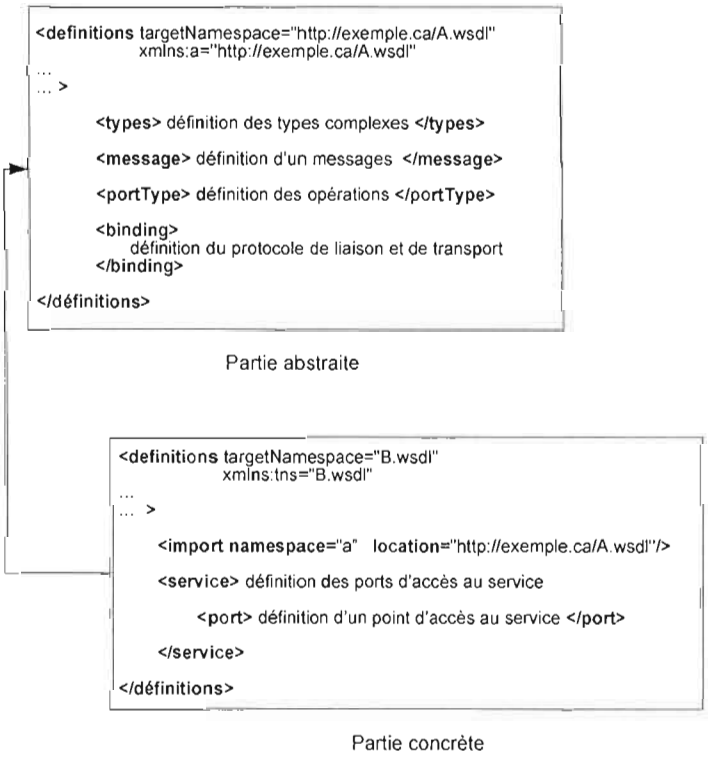

Figure 2.25. Séparation des éléments d'un document WSDL.

Un document WSDL décrit de façon détaillée comment utiliser un service Web. Cependant, il n'est pas toujours facile de trouver un service Web ou de localiser son document WSDL. Dans la section suivante, nous introduisons le registre UDDI qui résout ce problème.

2.4. UDDI (Universal Description, Discovery and Integration)

Proposé par un groupe d'entreprises qui œuvrent dans le domaine informatique, UDDI (OASIS-UDDI, 2002) est un annuaire qui permet aux entreprises fournissant des services Web de s'enregistrer et de publier les services Web qu'elles offrent au grand public. L'idée principale d'UDDI est de standardiser le format des entrées d'entreprise et de services dans un annuaire. Ceci permettant, éventuellement, d'automatiser le processus de découverte de services Web afin de faciliter les échanges d'affaires entre entreprises (B2B).

Un registre UDDI est défini comme un méta service qui utilise les protocoles SOAP et HTTP et qui suit un modèle d'échange de données basé sur le langage XML (Figure 2.26).

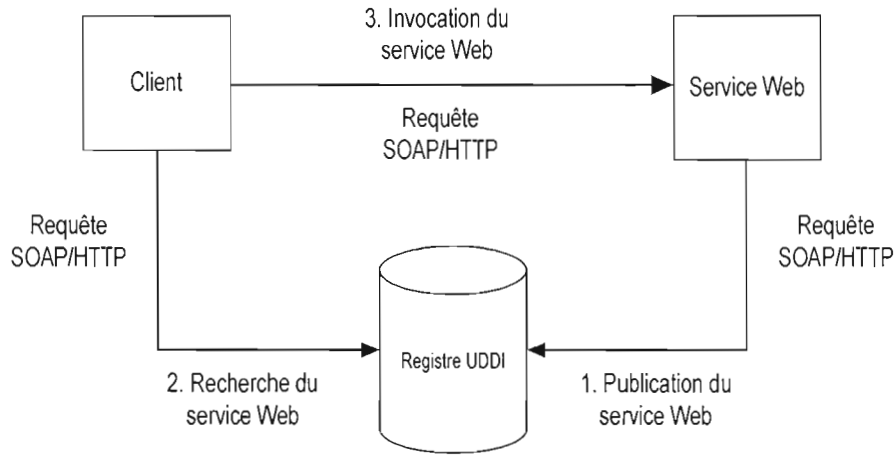

Figure 2.26. Publication et recherche dans un registre UDDI.

Actuellement, il existe sur le marché plusieurs implémentations de registres UDDI. Microsoft et IBM ont été les premiers à mettre sur le marché des registres UDDI accessibles à travers un navigateur Web. Ils fournissent aussi des registres UDDI d'essai pour encourager les entreprises à utiliser cette norme.

2.4.1. Structure de l'information dans un registre UDDI

Un enregistrement UDDI est un document XML qui décrit une entreprise et les services Web qu'elle offre. Il contient trois types d'informations :

- Page blanche : elle contient les détails d'une entreprise. Ces détails sont décrits dans une entité de type **`BussinessEntity`** (Figure 2.27).

- Page jaune : elle contient les détails d'un service Web proposé par une entreprise. Ces détails sont décrits dans une entité de type **`BussinessService`** (Figure 2.27).

- Page verte : elle contient les détails techniques d'un service Web et une référence vers une description détaillée de l'utilisation de ce service Web. Ces détails sont décrits dans des entités de type **`BindingTemplate`** et **`tModel`** (Figure 2.27).

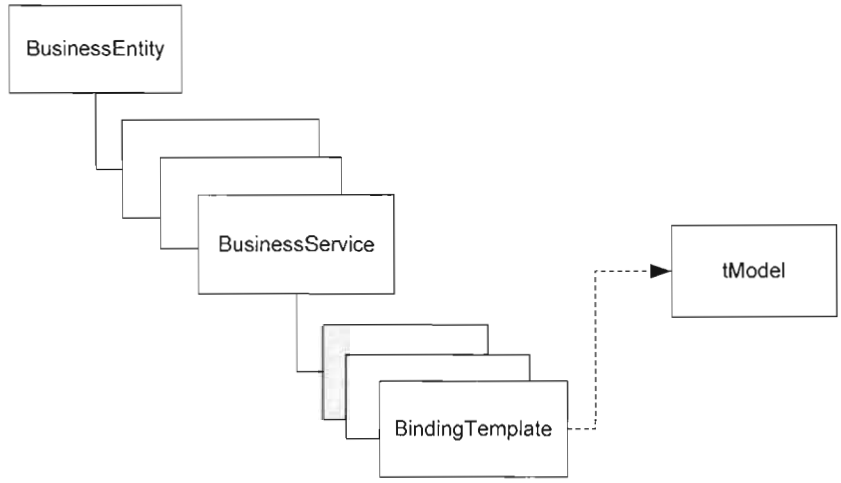

Figure 2.27. Structure de l'information dans un registre UDDI

2.4.2. Les requêtes UDDI

La spécification UDDI définit un ensemble de requêtes qui peuvent être exécutées sur un registre UDDI. Ces requêtes sont décrites dans un document WSDL. Un client qui veut envoyer une requête vers un registre UDDI commence par la formuler en respectant la description fournie dans le document WSDL. Par la suite, il met la requête dans une enveloppe SOAP. Enfin, il met l'enveloppe SOAP dans une requête HTTP puis l'envoie au registre UDDI (Figure 2.28). Chaque requête UDDI possède obligatoirement une réponse.

```
POST /registre HTTP/1.1
Host: www.exemple.ca
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "http//www.exemple.ca/registre"

<?xml version='1.0' ?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope">
      <S:Body>
            Requête UDDI
      </S:Body>
</S:Envelope>
```

Figure 2.28. Requête UDDI dans un message SOAP/HTTP

Une réponse (Figure 2.29) est construite de la même façon au niveau du registre avant d'être retournée au client.

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<?xml version='1.0' ?>
<S:Envelope xmlns:S=" " >
      <S:Body>
            Réponse UDDI
      </S:Body>
</S:Envelope>
```

Figure 2.29. Réponse UDDI dans un message SOAP/HTTP

Les requêtes UDDI sont réparties en deux groupes, i) les requêtes de publication qui permettent d'enregistrer, de modifier ou de supprimer des informations concernant une entreprise ou un service Web au niveau du registre, ii) les requêtes de recherche qui permettent de consulter les informations des entreprises ou des services Web enregistrés au niveau du registre. La Figure 2.30 décrit la liste complète des requêtes UDDI définies dans le document de spécification de la version 2.0.

Requêtes de publication

- **get_authToken :** retourne une clé d'authentification.
- **discard_authToken:** désactive une clé d'authentification.
- **save_business :** enregistre les détails d'une entreprise.
- **save_service :** enregistre les détails d'un service Web.
- **save_binding :** enregistre les détails des protocoles de liaison au service Web.
- **save_tModel :** enregistre les détails techniques d'un service Web.
- **delete_business :** supprime les détails d'une entreprise.
- **delete_service :** supprime les détails d'un servie Web.
- **delete_binding :** supprime les détails des protocoles de liaison au service Web.
- **delete_tModel :** supprime les détails techniques d'un service Web.

Requêtes de recherche

- **find_business** : cherche les entreprises qui vérifient certains critères.
- **find_service** : cherche les services Web qui vérifient certains critères.
- **find_binding** : cherche les protocoles de liaison qui vérifient certains critères.
- **find_tModel** : cherche les modèles techniques qui vérifient certains critères.
- **get_businessDetail** : retourne les détails d'une entreprise.
- **get_serviceDetail** : retourne les détails d'un service Web.
- **get_bindingDetail** : retourne les détails des protocoles de liaison d'un service Web.
- **get_tModelDetail** : retourne les détails techniques d'un service Web.

Figure 2.30. Liste des requêtes de publication et de recherche du registre UDDI

2.4.3. Publication d'un service Web dans un registre UDDI

Pour illustrer quelques modèles de requêtes de publication, voici un scénario qui décrit la publication des informations d'une entreprise et d'un service qu'elle offre. L'entreprise *X*, œuvrant dans le domaine du développement de logiciel, veut publier le service Web *Calculatrice* décrit dans la Figure 2.10 dans un registre UDDI.

Une requête de publication nécessite, tout le temps, une clé d'authentification. Cette clé peut être récupérée par une requête de type **`get_authToken`** (Figure 2.31)

```
<get_authToken generic="2.0" xmlns="urn:uddi-org:api_v2"
               userID="radhouane"
               cred="password" />
```

Figure 2.31. Requête pour demander une clé d'authentification

Si le client détient les privilèges de publication sur le registre UDDI, il reçoit une entité de type **authInfo** (Figure 2.32) qui contient une clé d'authentification qui va servir aux requêtes de publication.

```
<authInfo> e22a4f61e22a4f6</authInfo>
```

Figure 2.32. Clé d'authentification

Une fois que la clé d'authentification est reçue, le client prépare une requête de type **save_bussiness** (Figure 2.33) qui sert à enregistrer les informations de l'entreprise et l'envoie au registre UDDI.

```
<save_business generic="2.0" xmlns="urn:uddi-org:api_v2">
      <authInfo>e22a4f61e22a4f6</authInfo>
      <businessEntity businessKey="">
        <name xml:lang="fr"> X </name>
        <description xml:lang="fr">
                Entreprise de développement de logiciel
        </description>
        <contacts>
          <contact useType="">
            <personName>radhouane ben tamrout </personName>
            <phone>514 999 9999 </phone>
            <email>radouane56@hotmail.com</email>
          </contact>
        </contacts>
        <categoryBag>
          <keyedReference
               tModelKey="uuid:DB77450D-9FA8-45D4-A7BC-04411D"
               keyName="Remote access software"
               keyValue="43162705"/>
        </categoryBag>
      </businessEntity>
</save_business>
```

Figure 2.33. Requête de publication des informations d'une entreprise

Si tout va bien, le client reçoit, en réponse, une entité de type **businessDetail** (Figure 2.34) qui contient les informations de l'entreprise enregistrée et une clé d'identification unique de l'entreprise. Cette clé servira par la suite pour enregistrer les services Web offerts par cette entreprise.

```
<businessDetail generic="2.0" xmlns="urn:uddi-org:api_v2">
   <businessEntity businessKey="f58ad65d-53f5-8ad6-c196-edeb4">
        <name xml:lang="fr"> X </name>
        <description xml:lang="fr">
                Entreprise de développement de logiciel
        </description>
        <contacts>
          <contact useType="">
            <personName>radhouane ben tamrout </personName>
            <phone>514 999 9999 </phone>
            <email>radouane56@hotmail.com</email>
          </contact>
        </contacts>
        <categoryBag>
          <keyedReference
               tModelKey="uuid:DB77450D-9FA8-45D4-A7BC-04411D"
               keyName="Remote access software"
               keyValue="43162705"/>
        </categoryBag>
      </businessEntity>
</businessDetail>
```

Figure 2.34. Confirmation de publication des informations d'une entreprise

Une fois que les détails de l'entreprise sont enregistrés, on passe à l'étape de publication de service Web qu'elle offre. Alors, le client prépare une requête de type **`save_service`** (Figure 2.35) qui contient les détails d'un service Web.

```
<save_service xmlns="urn:uddi-org:api_v2" generic="2.0">
      <authInfo>88727f7e88727f7</authInfo>
      <businessService
           businessKey="f58ad65d-53f5-8ad6-c196-edeb4"
           serviceKey="">
           <name xml:lang="fr"> calculatrice </name>
           <description xml:lang="fr">
                Une calculatrice qui fait l'addition
           </description>
      </businessService>
</save_service>
```

Figure 2.35. Requête de publication des informations d'un service Web

Il reçoit, en réponse, une entité de type **`serviceDetail`** (Figure 2.36) qui contient les informations du service enregistré, et une clé d'identification unique du service Web.

```
<serviceDetail generic="2.0" operator=" "
               xmlns="urn:uddi-org:api_v2">
   <businessService
          businessKey="f58ad65d-53f5-8ad6-c196-edeb4"
          serviceKey="f12ad65d-53f5-8bt4-c14e-dfee7" >
          <name xml:lang="fr"> calculatrice </name>
          <description xml:lang="fr">
            Une calculatrice qui permet l'addition de deux entiers
          </description>
      </businessService>
</serviceDetail>
```

Figure 2.36. Confirmation de publication des informations d'un service Web

À ce stade, l'adresse du service Web et l'adresse de son document WSDL ne sont pas encore enregistrées au niveau du registre UDDI. Ces deux éléments représentent les détails techniques du service Web. La spécification UDDI considère le document WSDL, qui décrit la partie abstraite d'un service Web, comme un élément réutilisable puisqu'un service peut être publié plus d'une fois. La requête **`save_tModel`** (Figure 2.37) permet d'enregistrer des données réutilisables dans une entité de type **`tModel`**. Un **`tModel`** est défini comme étant une méta-donnée réutilisable.

```
<save_tModel xmlns="urn:uddi-org:api_v2" generic="2.0">
  <authInfo>833905c9833905c</authInfo>
  <tModel   tModelKey="f59aba30-c6f5-9aba-61a6-6fd2bfcdbb0c">
      <name> tModel du WSDL calculatrice</name>
      <description xml:lang="en">
         C'est un tModel qui fait référence au document WSDL
      </description>
      <overviewDoc>
         <overviewURL>
            http://exemple.com/calculatrice.wsdl
         </overviewURL>
      </overviewDoc>
      <categoryBag>
         <keyedReference
              tModelKey="C1ACF26D-9672-4404-9D70-39B756E62AB4"
              keyName="uddi-org:types" keyValue="wsdlSpec"/>
      </categoryBag>
  </tModel>
</save_tModel>
```

Figure 2.37. Requête de publication d'un modèle technique réutilisable

Le client reçoit, en réponse, une entité de type **`tModelDetail`** (Figure 2.38), qui contient les informations du **`tModel`** enregistré, et une clé d'identification unique pour le

tModel. L'enregistrement d'un **tModel** n'est pas nécessaire dans le cas où les détails techniques du service Web font référence à un **tModel** existant.

```
<tModelDetail generic="2.0" xmlns="urn:uddi-org:api_v2">
   <tModel   tModelKey="f59aba30-c6f5-9aba-61a6-6fd2b">
      <name> tModel du WSDL calculatrice</name>
      <description xml:lang="en">
         C'est un tModel qui fait référence au document WSDL
      </description>
      <overviewDoc>
         <overviewURL>
            http://exemple.com/calculatrice.wsdl
         </overviewURL>
      </overviewDoc>
      <categoryBag>
         <keyedReference
              tModelKey="ClACF26D-9672-4404-9D70-39B756E62AB4"
              keyName="uddi-org:types" keyValue="wsdlSpec"/>
      </categoryBag>
   /tModel>
</tModelDetail>
```

Figure 2.38. Confirmation de publication d'un modèle technique réutilisable

Une fois que le **tModel** est créé, le client envoie une requête de type **save_binding** (Figure 2.39) au registre. Cette requête enregistre l'adresse du service Web et une référence vers le **tModel** qui pointe sur l'adresse du document WSDL du service.

```
<save_binding xmlns="urn:uddi-org:api_v2" generic="2.0">
   <authInfo>87e9f26387e9f26</authInfo>
   <bindingTemplate bindingKey=""
                    serviceKey="f12ad65d-53f5-8bt4-c14e-dfee7">
        <description>
             Les détails technique du service calculatrice
        </description>
        <accessPoint URLType="http">
              http://exemple.com/calculatrice
        </accessPoint>
        <tModelInstanceDetails>
             <tModelInstanceInfo
                tModelKey="f59aba30-c6f5-9aba-61a6-6fd2b"/>
        </tModelInstanceDetails>
   </bindingTemplate>
</save_binding>
```

Figure 2.39. Requête de publication des informations techniques d'un service Web

Il reçoit, en réponse, une entité de type **bindingDetail** (Figure 2.40) qui contient les informations techniques du service Web.

```
<bindingDetail generic="2.0" operator=" "
               xmlns="urn:uddi-org:api_v2">
   <bindingTemplate bindingKey="f58baeld-2af5-8bae-led6-0c3f"
                    serviceKey="f12ad65d-53f5-8bt4-c14e-dfee7">
        <description>
             Les détails techniques du service calculatrice
        </description>
        <accessPoint URLType="http">
              http://www.exemple.com/calculatrice
        </accessPoint>
        <tModelInstanceDetails>
             <tModelInstanceInfo
                tModelKey="f59aba30-c6f5-9aba-61a6-6fd2b"/>
        </tModelInstanceDetails>
   </bindingTemplate>
</bindingDetail>
```

Figure 2.40. Publication des informations techniques d'un service Web

La figure (Figure 2.41) décrit le scénario complet de publication des informations de l'entreprise et des services Web qu'elle offre.

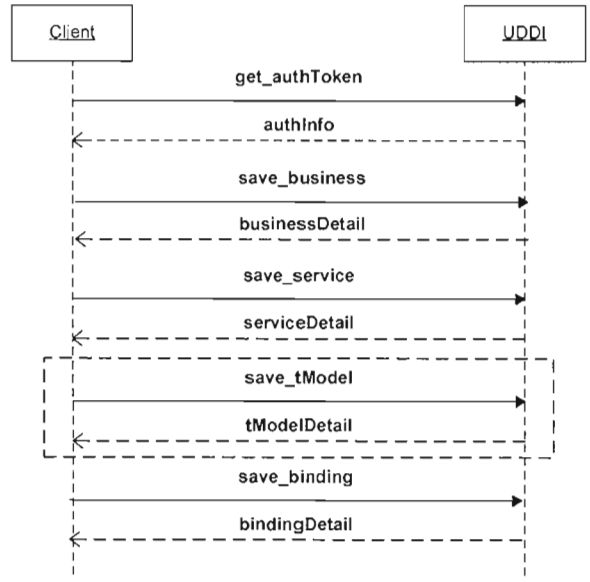

Figure 2.41. Scénario de publication

2.4.4. Recherche d'un service Web dans un registre UDDI

La spécification UDDI définit une dizaine de requêtes de recherche sur un registre. Ces requêtes permettent aux utilisateurs de chercher les informations d'une entreprise ou d'un service Web selon plusieurs critères et sur plusieurs niveaux. Voici un scénario qui décrit la recherche du service Web *calculatrice* (Figure 2.10) offert par l'entreprise *X*.

Le client envoie une requête **`find_business`** (Figure 2.42) qui cherche toutes les entreprises ayant le nom *X*. Il est possible de trouver plus d'une entreprise ayant le même nom sur un registre UDDI.

```
<find_business generic='2.0' xmlns='urn:uddi-org:api_v2'>
        <name> X </name>
</find_business>
```

Figure 2.42. Requête de recherche d'une entreprise

Il reçoit en réponse une entité de type **`businessList`** (Figure 2.43) qui contient la liste des entreprises ayant le nom *X* et la liste des services Web qu'elles offrent.

```
<businessList xmlns="urn:uddi-org:api_v2" generic="2.0" operator="">
   <businessInfos>
      <businessInfo businessKey="f58ad65d-53f5-8ad6-c196-edeb4">
         <name xml:lang="fr"> X </name>
         <description xml:lang="fr">
              Entreprise de développement de logiciel
         </description>
         <serviceInfos>
            <serviceInfo serviceKey="f12ad65d-53f5-8bt4-c14e-dfee7"
                         businessKey="f58ad65d-53f5-8ad6-c196-edeb4">
               <name xml:lang="fr"> calculatrice </name>
            </serviceInfo>
         </serviceInfos>
      </businessInfo>
   </businessInfos>
</businessList>
```

Figure 2.43. Résultat de la recherche d'une entreprise

Le client choisit le service Web qu'il veut consulter. Il envoie une requête de type **`get_serviceDetail`** (Figure 2.44) qui contient une clé unique. Chaque service Web enregistré au niveau du registre UDDI possède une clé unique.

```
<get_serviceDetail generic="2.0" xmlns="urn:uddi-org:api_v2" >
      <serviceKey> f12ad65d-53f5-8bt4-c14e-dfee7 </serviceKey>
</get_serviceDetail>
```

Figure 2.44. Requête de recherche d'un service Web

Il reçoit en réponse une entité de type **serviceDetail** (Figure 2.45) qui contient en détail des informations sur le service Web, comment y accéder et comment l'utiliser.

```
<serviceDetail xmlns="urn:uddi-org:api_v2"
               generic="2.0" operator="">
   <businessService serviceKey="f12ad65d-53f5-8bt4-c14e-dfee7"
                    businessKey="f58ad65d-53f5-8ad6-c196-edeb4">
      <name>calculatrice</name>
      <description xml:lang="fr">
         Une calculatrice qui fait l'addition
      </description>
      <bindingTemplates>
        <bindingTemplate bindingKey="f58baeld-2af5-8bae-1ed6-0c3f"
                         serviceKey=" f12ad65d-53f5-8bt4-c14e-dfee7">
           <description>
             Les détails techniques du service calculatrice
           </description>
           <accessPoint URLType="http">
                http://exemple.com/calculatrice
           </accessPoint>
           <tModelInstanceDetails>
                <tModelInstanceInfo
                       tModelKey="f59aba30-c6f5-9aba-61a6-6fd2b"/>
           </tModelInstanceDetails>
        </bindingTemplate>
      </bindingTemplates>
   </businessService>
</serviceDetail>
```

Figure 2.45. Résultat de la recherche d'un service Web

Le document WSDL de ce service Web peut être récupéré en envoyant une requête de type **get_tModelDetail** (Figure 2.46). Cette requête doit contenir une clé unique du **tModel** qui fait référence au document WSDL de ce service Web.

```
<get_tModelDetail generic="2.0">
<tModelKey> f59aba30-c6f5-9aba-61a6-6fd2b </tModelKey>
</get_tModelDetail>
```

Figure 2.46. Requête de recherche d'un modèle technique

Le client reçoit en réponse une entité de type **tModelDetail** (Figure 2.47) qui contient une référence vers l'URL du document WSDL de ce service

```
<tModelDetail generic="2.0" operator=" "
                xmlns="urn:uddi-org:api_v2">
   <tModel   tModelKey="f59aba30-c6f5-9aba-61a6-6fd2b">
      <name> tModel du WSDL calculatrice</name>
      <description xml:lang="en">
         C'est un tModel qui fait référence au document WSDL
      </description>
      <overviewDoc>
         <overviewURL>
            http://exemple.com/calculatrice.wsdl
         </overviewURL>
      </overviewDoc>
      <categoryBag>
         <keyedReference
              tModelKey="C1ACF26D-9672-4404-9D70-39B756E62AB4"
              keyName="uddi-org:types" keyValue="wsdlSpec"/>
      </categoryBag>
   /tModel>
</tModelDetail>
```

Figure 2.47. Réponse de la recherche d'un modèle technique

À ce niveau, le client détient toute l'information nécessaire pour invoquer le service Web en question. La Figure 2.48 décrit le scénario complet de recherche.

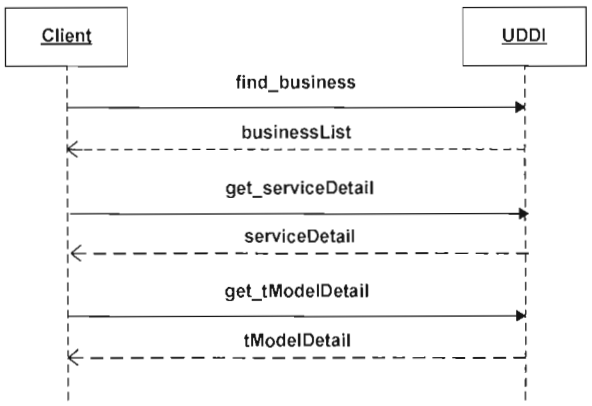

Figure 2.48. Scénario de recherche

Grâce aux registres UDDI, les entreprises espèrent augmenter leurs échanges inter-entreprises. En théorie, les registres UDDI doivent permettre aux applications de trouver automatiquement les services Web dont ils ont besoin pour ensuite les invoquer. Cependant, la pratique est encore loin de ça.

2.5. Conclusion

Les Services Web s'annoncent plus que jamais comme la solution optimale aux problèmes d'échanges de données et d'intégration d'applications. Malgré les efforts qui ont été mis en œuvre pour donner naissance au protocole SOAP, au descripteur WSDL et aux annuaires UDDI, il reste beaucoup à faire. En effet, l'aspect sécurité est presque inexistant et la performance est faible en la comparant avec celle des approches telles RMI et CORBA

CHAPITRE III

Recherche de partenaires dans un registre UDDI

Comme nous venons de le voir, la norme UDDI propose, 1) un format standard pour la description d'entreprises et de services, 2) un ensemble de requêtes standards pour publier de l'information dans un registre, et pour consulter un registre. Comme toute norme industrielle, la norme UDDI est un compromis entre idéalisme et pragmatisme. Idéalement, on aimerait que la description d'entreprises et de services soit riche pour inclure le maximum d'informations dont on peut avoir besoin sur un partenaire potentiel en affaires, ou sur les services qu'il offre. La norme se doit aussi d'être pragmatique dans le sens où elle doit être facile à mettre en œuvre pour que les entreprises y adhèrent. Cela veut dire, au contraire, que le type de descriptions de services que l'on doit publier doit être facilement accessible, et ne doit pas impliquer un investissement important de la part des entreprises au niveau de la spécification ou de la formation de personnel. En fait, la norme UDDI ne spécifie *même pas* de langage de description de services, en tant que tels, pour laisser libre choix aux entreprises de publier la description de leurs services dans le langage qui est le plus approprié pour leur domaine. Cela étant, il y a quasiment une norme de facto : l'utilisation de descriptions WSDL.

Les utilisateurs sophistiqués de registres, et les chercheurs en quête de problèmes à résoudre, ont identifié un certain nombre d'utilisations de registres pour lesquelles la norme UDDI s'avère inadéquate. Ils ont alors proposé d'augmenter, et 1) la richesse des descriptions de services, et 2) les fonctionnalités de consultations du registre. Dans ce chapitre, nous passons en revue les différents problèmes à résoudre pour lesquels la norme UDDI est insuffisante. Par la suite, nous décrirons, pour chaque problème, les solutions suggérées dans la littérature.

3.1. Problèmes

3.1.1. Problème de sémantique

Les techniques actuelles ne permettent pas à des services Web d'interagir de manière intelligente en étant capable de se découvrir automatiquement, de négocier entre eux et de se composer dynamiquement en des services plus complexes. Une telle composition permettra de réaliser des opérations plus ou moins complexes. Comme exemple de ces opérations complexes de la vie quotidienne ou professionnelle, on pourra citer l'organisation d'un voyage ou la souscription d'un contrat d'assurance. Des standards comme WSDL et UDDI ne décrivent pas suffisamment les connaissances de manière à les rendre exploitables par des machines. Il est donc nécessaire d'intégrer une sémantique qui permet de représenter et de manipuler des connaissances non seulement structurelles, mais aussi de nature fonctionnelle. Dans une telle optique, la recherche et la découverte de services Web seront dotées de capacités de raisonnement qui s'appuient sur la description formalisée et sur la mise en relation des différentes sources d'information. De ce fait, la recherche de services se fera non seulement par mots-clés, mais aussi, selon le contexte dans lequel ils doivent être compris et les différents concepts qui leurs sont associés.

3.1.2. Problème de qualité de service (QoS)

Découvrir un service Web qui réponde à un besoin n'est pas suffisant. Diverses propriétés de qualité de service telles que la disponibilité, la performance et la fiabilité sont nécessaires pour effectuer le bon choix de service Web. Par exemple, dans le scénario décrit dans la section précédente, une recherche sur un registre UDDI standard, permettrait de découvrir tous les services Web qui accèdent aux tarifs et aux disponibilités des billets d'avion de plusieurs compagnies aériennes. Cette recherche donne une description de tous les services Web découverts, mais ne donne aucune idée sur le temps de réponse ni sur la disponibilité de ces services. Il devient donc nécessaire de pouvoir extraire et exploiter les services Web pertinents parmi les nombreux collectés. Dans un contexte de découverte dynamique de partenaires d'affaires, il est important d'en trouver le meilleur. C'est important de prioriser la recherche sur les registres UDDI surtout dans un monde où il y a une augmentation rapide du nombre de services Web offerts par les entreprises.

3.1.3. Problème de composition dynamique de service Web

La composition dynamique de services Web est en quelque sorte la création de nouvelles fonctionnalités en combinant plusieurs fonctionnalités offertes afin de réaliser une tâche précise. Elle implique la capacité de sélectionner, de composer et de faire interagir des services existants dynamiquement selon les besoins. Cette composition n'est pas exécutée de manière prévisible et répétitive. UDDI ne permet pas d'assembler et d'orchestrer des services Web. Il ne propose pas de solutions de composition et d'adaptation dynamique de services. La description des services Web que UDDI propose n'intègre que très peu l'aspect sémantique. Ceci ne permet pas de garantir que la composition fonctionnelle des services Web soit correcte à l'exécution.

3.2. Classification des approches étudiées

Durant notre étude et évaluation des travaux existants et en nous basant sur une classification par problème résolu, nous avons pu identifier trois familles d'approches qui apportent des solutions aux problèmes décrits ci-dessus :

1. Prise en compte de la sémantique des services Web : les descriptions existantes se limitent à la signature des services. Il s'agit ici d'augmenter de telles descriptions, et les fonctionnalités de découvertes correspondantes, pour prendre en compte la sémantique des services.

2. Prise en compte de la qualité de service : Il s'agit ici d'explorer les possibilités pour augmenter la description des fonctionnalités des services web — qu'elle soit syntaxique ou sémantique — par des critères de qualité de service.

3. La composition dynamique de services Web : dans le cas où un besoin fonctionnel ne peut être satisfait par un seul service, mais qu'il existe une composition de services pour répondre au besoin, on aimerait que le registre UDDI soit capable de découvrir ces compositions, et de les mettre en œuvre à l'exécution.

Dans ce qui suit, nous allons voir, en détail, les trois familles d'approches décrites ci-dessus. Dans la section 3.3, nous présentons quelques approches qui visent une extension sémantique des descriptions de services. La section 3.4 traite des extensions visant l'intégration de critères de qualité de services. La section 3.5 traite le problème de composition dynamique de services Web. Dans la section 3.6, nous discutons des avantages et des inconvénients des approches présentées et nous introduisons la motivation pour laquelle nous proposons une nouvelle approche.

3.3. Prise en compte de la sémantique dans la découverte de services Web

Plusieurs approches ont traité l'aspect sémantique lié aux services Web. Ces approches partagent le but commun de rendre le processus de découverte et d'invocation de service Web plus accessible aux machines, et cela, pour atteindre de bons niveaux d'automatisation. Certaines recherches ont proposé d'ajouter une couche sémantique à WSDL et à UDDI pour répondre aux limites de ces derniers (Sivashanmugam et al., 2003), (Paolucci et al., 2003), (Srinivasan et al., 2004), (Colgrave et al., 2004), (Akkiraju et al., 2003) et (Pilioura et al., 2003). L'idée partagée par ces recherches est de décrire le « quoi? » et le « pourquoi? », et non seulement le « comment? ». Nous nous intéressons dans notre recherche aux approches qui intègrent l'aspect sémantique au processus de découverte de services Web assuré par les registres UDDI. Nous avons choisi, parmi les approches que nous avons étudiées, deux approches qui nous semblent les plus intéressantes. Srinivasan et al. (Srinivasan et al., 2004) proposent une approche qui ajoute de l'information sémantique sous format OWL-S (OWL-S, 2003) dans un registre UDDI afin d'augmenter la capacité de recherche et de découverte de services Web. Colgrave et al. (Colgrave et al., 2004) proposent une approche qui intègre un service de correspondance sémantique externe dans le processus de découverte de service Web. Ceci, en gardant intact, la structure du registre UDDI standard. Dans ce qui suit, nous allons voir en détail, les deux approches décrites au-dessus.

3.3.1. Ajout de OWL-S au registre UDDI

Selon Srinivasan et al., le mécanisme de découverte de services Web est très limité. UDDI permet de garantir une description syntaxique de la façon d'interagir avec un service

Web, mais ne donne aucune description sémantique de ce qu'il fait, ou comment il le fait. Deux services Web qui ont la même description syntaxique peuvent ne pas faire la même chose et vice-versa (Srinivasan et al., 2004). Plusieurs initiatives ont essayé de résoudre ce problème. Des langages comme RDF et OWL, basés sur XML, sont apparus pour faire face à cette problématique. Dans cette approche, Srinivasan et al. utilisent l'ontologie OWL-S (OWL-S, 2003) - nouveau nom pour DAML-S - afin de décrire un service Web et d'exprimer ces fonctionnalités. Cette ontologie décrit les entrées, les sorties et les pré-conditions nécessaires au bon fonctionnement d'un service Web (Srinivasan et al., 2004). OWL-S décrit un service Web en fonction de ce qu'il fait. Ceci, dans le but de permettre aux utilisateurs de voir s'il leur convient ou non. Selon Srinivasan et al., la combinaison de OWL-S avec les registres UDDI permet d'augmenter la capacité de recherche et de découverte des services Web. En combinant les deux, nous tirons profit de la popularité d'UDDI et de la richesse sémantique fournie par OWL-S (Srinivasan et al., 2004).

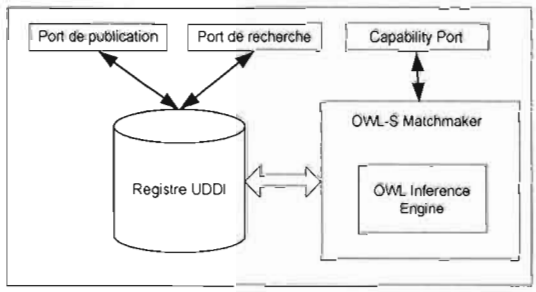

Figure 3.1. Architecture OWL-S / UDDI. Tirée de (Srinivasan et al., 2004)

Srinivasan et al. proposent une architecture (Figure 3.1) qui permet de stocker des informations sémantiques propres à OWL-S dans un registre UDDI afin d'augmenter les possibilités de recherche sur ce dernier (Srinivasan et al., 2004). Cette architecture, appelée '*OWL-S / UDDI Matchmaker Architecture*', est composée d'un registre UDDI standard et d'un *OWL-S matchmaker* qui offre une interface '*capability port*' permettant de faire des recherches de services Web en se basant sur des informations sémantiques (i.e. entrées, sorties, prés conditions, etc.).

Une requête de publication de service Web, reçue par le registre UDDI de cette architecture, est traitée de la même façon que sur un registre UDDI standard. Le service Web

est publié sur le registre. Si ce dernier découvre que cette requête contient des informations sémantiques propres à un *OWL-S*, il l'envoie au OWL-S matchmaker. Ce dernier classifie le service Web en se basant sur les informations sémantiques reçues dans la requête.

Les recherches de service Web sur ce modèle sont assurées par un ensemble d'APIs qui étendent les APIs standards d'UDDI. Un client qui veut faire une recherche de services Web, en se basant sur des informations sémantiques, envoie une requête de recherche à l'interface 'capability port'. Le composant OWL-S matchmaker reçoit cette requête puis fait une correspondance sémantique entre celle-ci et les services Web offerts. La réponse à cette requête contient une liste de clés de services Web et le niveau de correspondance entre chaque service Web trouvé et la requête du client. Ces informations permettent de sélectionner et d'invoquer le service Web qui répond le mieux aux exigences du client (Srinivasan et al., 2004).

Nous allons voir, dans ce qui suit, le modèle adopté dans cette approche pour stocker les informations propres à OWL-S dans un registre UDDI. Ensuite, nous présenterons l'algorithme de correspondance sémantique utilisé par le matchmaker pour faire la correspondance sémantique entre une requête de recherche et un ensemble de services Web publiés sur un registre UDDI.

a) Modèle de stockage de OWL-S dans un registre UDDI

L'approche proposée par Srinivasan et al. utilise un modèle de correspondance entre OWL-S et UDDI. Ce modèle est basé sur un mécanisme de correspondance d'attributs un à un entre un *OWL-S Profile* et les entités **`businessEntity`** et **`businessService`** de UDDI (Figure 3.2). Les attributs de *OWL-S Profile* qui n'ont pas de correspondant au niveau d'UDDI sont définis dans des entités de type **`tModel`**. Un **`tModel`** est créé pour chaque attribut de *OWL-S Profile* qui n'est pas défini dans UDDI. Ensuite, on fait référence à ces **`tModels`** au niveau du **`businessService`** (Figure 3.2). Les APIs de publication UDDI ont été étendues pour réaliser ce type de correspondance avant de publier un service Web.

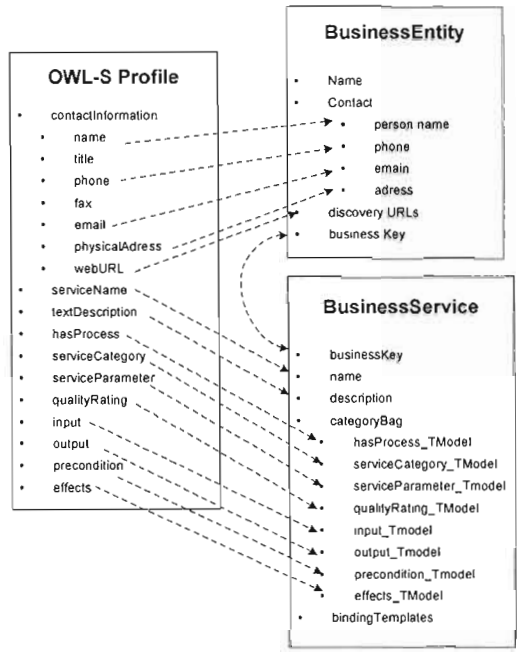

Figure 3.2. Modèle de correspondance OWL-S/UDDI. Tirée de (Srinivasan et al., 2004)

Un *OWL-S Profile* (Figure 3.2) est une composante du *OWL-S* qui fournit une description détaillée d'un service Web et de son fournisseur. La description du service Web donne une idée sur le comportement fonctionnel de ce dernier : elle définit l'ensemble d'entrées, l'ensemble de sorties, et l'ensemble de pré-conditions nécessaires à son bon déroulement. Elle contient aussi un ensemble d'attributs utiles à la sélection automatique d'un service Web tels que la performance et les contraintes liées à un service Web (Martin et al., 2003).

b) Algorithme de correspondance sémantique

Le composant *'OWL-S matchmaker'* utilise un algorithme de correspondance sémantique pour faire la recherche de service Web (Srinivasan et al., 2004). Quand une requête de recherche est reçue, l'algorithme commence par faire correspondre les sorties de services Web demandés dans la requête avec toutes les sorties des services Web publiés. Le résultat de cette correspondance donne un ensemble de services Web. Par la suite, l'algorithme fait correspondre les entrées de services Web demandés dans la requête avec les entrées de tous les services Web sélectionnés à l'étape précédente. Le résultat final est

l'ensemble des services Web qui possèdent les entrées et les sorties correspondantes à celle demandée dans la requête. Le degré de correspondance entre une sortie - ou une entrée - demandée dans la requête et une sortie - ou une entrée – assurée par un service Web publié dépend du niveau de correspondance entre les deux concepts qui les définissent. L'algorithme de correspondance ne fait pas de comparaison syntaxique entre concepts, il se base sur la relation entre ces concepts au sein d'une ontologie OWL. Prenons l'exemple d'un service Web de vente de véhicule. Ce service peut spécifier comme élément de sortie '*véhicule*' et un demandeur de service Web peut spécifier dans sa requête de recherche l'élément '*voiture*' comme sortie. Syntaxiquement il n'y a pas de correspondance entre le concept '*véhicule*' et le concept '*voiture*' mais du point de vue ontologique le concept '*véhicule*' inclue dans une classification le concept '*voiture*' (Srinivasan et al., 2004). La Figure 3.3 montre le lien entre ces deux concepts. Une recherche de services Web basée sur cet algorithme retourne tous les services Web qui possèdent une sortie liée ontologiquement avec la sortie demandée.

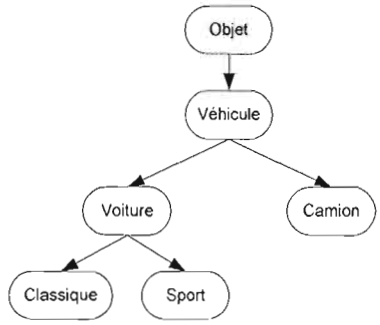

Figure 3.3. Ontologie d'un véhicule. Tirée de (Srinivasan et al., 2004)

L'algorithme proposé dans cette approche (Srinivasan et al., 2004) définit quatre niveaux de correspondance entre un concept demandé et les concepts existants. Supposons que le concept demandé est une sortie. *sortieD* représente la sortie demandée dans une requête de recherche et *sortieE* représente la sortie d'un service Web déjà publié. Voici les quatre niveaux de correspondance définis par l'algorithme :

a. Si *sortieD* et *sortieE* sont identiques ou si *sortieD* est l'enfant immédiat de *sortieE* alors le niveau de correspondance est *'exact'*. Il représente le niveau de correspondance le plus haut.

b. Si *sortieD* est un concept qui représente une sous-classe de *sortieE* alors le niveau de correspondance est *'plug in'*. Il n'y a rien qui garantisse que *sortieD* sera présent. Par exemple, supposons que *sortieD* est représentée par le concept '*sport*' et *sortieE* est représentée par le concept '*véhicule*' (Figure 3.3). Un service Web qui possède une sortie 'véhicule' ne garantit pas nécessairement une sortie '*sport*'.

c. Si *sortieD* est une classe qui englobe *sortieE* alors le niveau de correspondance est *'subsume'*. Un service Web qui possède comme sortie *sortieE* n'est pas nécessairement satisfaisant. Par exemple, supposons que *sortieD* est représentée par le concept '*véhicule*' et *sortieE* est représenté par le concept '*sport*' (Figure 3.3). Un service Web qui possède une sortie '*sport*' ne garantit pas nécessairement une sortie '*classique*' ou '*camion*'.

d. Si *sortieD* et *sortieE* ne possèdent aucune relation alors le niveau de correspondance est *'fail'*. Aucun lien ontologique entre les deux concepts qui représentent *sortieD* et *sortieE*.

Dans la section suivante, nous verrons comment déterminer le niveau de correspondance entre un service Web et l'ensemble des requêtes possibles. Par la suite, nous expliquerons comment se fait la sélection de services Web au niveau du matchmaker.

c) Publication et recherche

Srinivasan et al. proposent de définir le niveau de correspondance entre un service Web et les requêtes possibles au moment de la publication de ce dernier. De son point de vue, le temps de publication d'un service Web n'est pas considéré comme critique. Donc, il est plus important d'exploiter ce temps pour réaliser la tâche de correspondance (Srinivasan et al., 2004). Cette tâche est assurée par le matchmaker au moment de la publication du service et non pas au moment de la recherche d'un service.

Dans ce modèle, le composant OWL-S matchmaker intervient au moment de la publication d'un service Web. Il maintient une taxonomie qui décrit les relations entre un ensemble de concepts. Cette taxonomie est mise à jour chaque fois qu'un nouveau service Web est publié. Le matchmaker charge les nouvelles ontologies qui définissent les entrées et les sorties du nouveau service Web et met à jour sa taxonomie en appliquant l'algorithme de correspondance présenté dans la section précédente. Chaque concept de cette taxonomie définit deux listes : '*output_node_information*' et '*input_node_information*'. Ces deux listes spécifient respectivement le niveau de correspondance entre le concept lui-même et les autres concepts qui représentent soit une entrée soit une sortie d'un service Web déjà publié. Par exemple, '*output_node_information*' définit le niveau de correspondance entre un concept pointé dans une requête de recherche et les autres concepts qui représentent les sorties des services Web existants. Cette liste sert à mettre en place une file de priorité des services Web qui peuvent être retournés dans le cas où une requête pointe sur ce concept. Elle est sous forme d'un vecteur [*<serv1,exact>,<serv2,subsume>*,.....] où *serv1* est un élément qui pointe sur un service Web et *exact* est le niveau de correspondance entre le concept lui-même et un autre concept qui représente une sortie de ce service (Srinivasan et al., 2004).

Le processus de recherche est moins coûteux en terme de temps. Pour chaque requête de recherche reçue, le matchmaker cherche les *output_node_informations* des concepts qui correspondent à chacune des sorties demandées dans la requête de recherche. Prenons l'exemple d'une requête de recherche qui contient les concepts '*voiture*' et '*prix*' comme éléments de sortie d'un service Web demandé. Le matchmaker extrait les *output_node_informations* des concepts '*voiture*' et '*prix*'. L'étape suivante consiste en la recherche de l'intersection entre ces deux listes. Si l'intersection est nulle alors il n'existe pas de service Web qui possède de telles sorties. Sinon le résultat est une liste de services Web dont chacun est associé à un niveau de correspondance. De même que pour les sorties, le matchmaker extrait les *input_node_informations* des entrées demandées puis effectue l'intersection entre les services Web qui ont été sélectionnés auparavant et les services Web présents dans les *input_node_informations*. Le résultat final est une liste de services Web ordonnée par niveau de correspondance (Srinivasan et al., 2004).

Pour implanter cette approche, Srinivasan et al. ont étendu une implémentation ouverte du registre UDDI réalisé par la fondation Apache (jUDDI, 2003). Les expérimentations préliminaires du registre OWL-S/UDDI (OWL-S/UDDI, 2004) ont montré que la publication d'un service Web sur ce dernier prend entre 6 à 7 fois plus de temps que sur un registre UDDI standard.

3.3.2. Modèle de correspondance externe

Colgrave et al. proposent un modèle qui permet d'augmenter la capacité de recherche sur un registre UDDI sans modifier la spécification de ce dernier (Colgrave et al., 2004). Ce modèle est une combinaison d'un registre UDDI avec des services externes permettant de faire une correspondance sémantique entre une requête client et les descriptions des services Web publiés sur ce registre. La Figure 3.4 montre le scénario complet de publication et de recherche de services Web du modèle proposé (Colgrave et al., 2004). Les fournisseurs publient l'adresse de la description du service qu'elles offrent. Les clients aussi publient l'adresse de la description du service demandé dans un `tModel` au niveau du registre UDDI et ce pour une réutilisation future. Ces deux descriptions sont assurées par des langages de description tels XML, DAML-S ou UML.

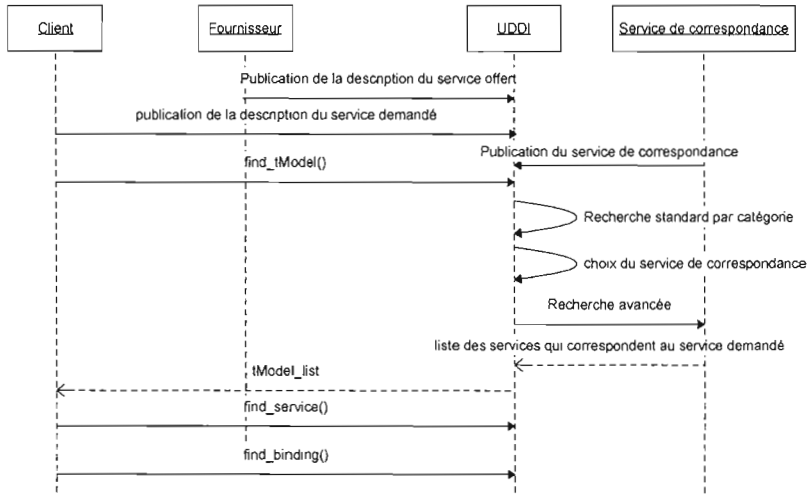

Figure 3.4. Scénario d'usage du modèle proposé par Colgrave et al.

Selon ce modèle, un client qui veut faire une recherche de services Web envoie une requête **`find_tModel()`** au registre UDDI. Cette requête doit indiquer qu'une correspondance externe est demandée. Une requête de recherche reçue par un registre UDDI est traitée en deux étapes. Une recherche standard est faite sur le registre UDDI et le résultat est envoyé à un service de correspondance externe qui va affiner cette recherche. Le registre UDDI choisit dynamiquement le service de correspondance selon le format de description utilisé. Plusieurs autres critères peuvent aussi être utilisés. Le résultat final est un ensemble de **`tModels`** ordonnés par niveau de correspondance entre services demandés et services offerts. Le client choisit un **`tModel`**, généralement le premier **`tModel`** de la liste retournée. Il envoie ensuite une requête standard **`find_service()`** et une requête **`find_binding()`** qui contiennent la clé du **`tModel`** choisi (Colgrave et al., 2004). Ces deux requêtes servent à trouver les informations nécessaires pour invoquer le service Web choisi.

a) Publication de la description d'un service Web

La publication de la description d'un service Web offert ou d'un service Web demandé au niveau du registre UDDI est assurée par des entités de type **`tModel`**. L'approche définit une nouvelle catégorie de **`tModel`**, la catégorie '*DescribedUsing*' qui caractérise l'ensemble des descriptions de services Web. De même, chaque type de description (DAML-S, UML, etc.) possède son propre **`tModel`** qui le catégorise. Pour différencier entre la description d'un service offert et la description d'un service demandé, l'approche utilise une catégorisation supplémentaire '*ClientRequirementsCategorizationTModel*' pour les **`tModels`** qui pointent sur la description d'un service demandé. Cette catégorisation est nécessaire pour le processus de recherche de service Web.

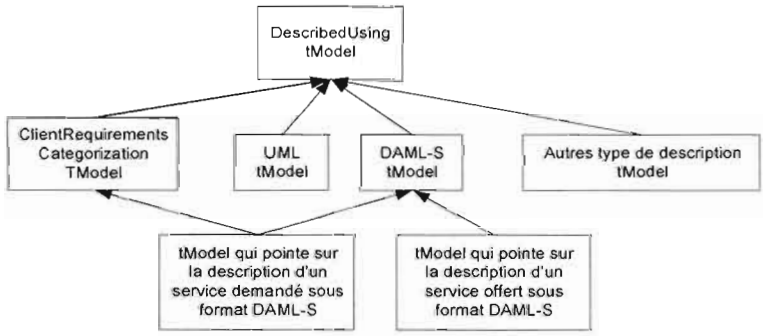

Figure 3.5. Catégorisation de la description pointée par un tModel

b) Publication des services de correspondance externes

Un service de correspondance est en réalité un service Web. De ce fait, un fournisseur qui veut offrir un service de correspondance publie ce dernier comme étant un service Web au niveau du registre UDDI. Cette technique permet l'intégration de plusieurs services de correspondance au registre UDDI. Par exemple, une description de service Web sous forme DAML-S nécessite un service de correspondance qui comprend le langage DAML-S, alors qu'une autre écrite en UML nécessite un service de correspondance qui comprend le langage UML. Donc, il est facile d'intégrer de nouveaux services de correspondance au fur et à mesure qu'un nouveau type de description est utilisé. Un fournisseur qui veut offrir un service de correspondance doit tout d'abord écrire son service en se conformant à un document WSDL qui standardise les interactions entre un registre UDDI et un service de correspondance. Ensuite il publie son service en suivant le processus standard de publication de service Web sur un registre UDDI. Chaque service de correspondance publié au niveau du registre doit faire référence à un **`tModel`** de catégorie '*ExternalMatchingService*' qui pointe sur le document WSDL décrivant ce service. Ceci permet de différencier les services de correspondances publiées sur le registre des autres services Web (Colgrave et al., 2004).

c) Recherche de service Web basée sur les besoins d'un client

La recherche de service Web selon cette approche diffère un peu du processus standard de recherche. Un client qui veut faire une recherche de service Web envoie une requête de type **`find_tModel()`** basé sur la catégorisation '*ClientRequirementsCategorizationTModel*' pour indiquer au registre UDDI que l'intervention d'un service de correspondance externe est requise dans le processus de recherche, et une référence vers le **`tModel`** qui pointe sur la description du service Web demandé. Cette requête peut contenir aussi d'autres types de catégorisation standard propre à UDDI.

```
<find_tModel generic="2.0" xmlns= "urn:uddi-org:api_v2">
   <categroyBag>
     <keyedReference tModelKey ="uuid:YYYYYYYY-YYYY-YYYY-YYYYYYYYYY"
                     keyName ="describedUsing"
                     keyValue ="uuid:CCCCCCCC-CCCC-CCCC-CCCCCCCCCC"/>
   </categoryBag>
</find_tModel>
```

Figure 3.6. Requête de type find_tModel

La Figure 3.6 montre un exemple de requête de type **find_tModel()**. L'attribut **tModelKey** fait référence à la catégorisation décrite au dessus, et l'attribut **keyName** fait référence au **tModel** qui pointe sur la description du service Web demandé.

Quand le registre UDDI reçoit une requête de type **find_tModel()**, il fait une première recherche en se basant sur le type de description du service Web demandé et sur les autres critères de catégorisation. Par exemple, un client écrit une requête **find_tModel()** qui fait référence à un **tModel** qui lui pointe sur une description du service Web demandé écrite sous format DAML-S. Il ajoute aussi dans sa requête une catégorisation standard NAICS1997 (UDDI-Taxonomy, 2004). Une première recherche au niveau du registre UDDI retourne tous les **tModels** qui pointent sur les descriptions de service Web du même format et qui font partie de la catégorisation NAICS1997. L'étape suivante consiste à choisir le service de correspondance qui convient le mieux. Le registre UDDI fait une sélection du service de correspondance en se basant sur le type de la description du service Web demandé. A titre d'exemple, si la description du service Web demandé est sous forme DAML-S alors le registre UDDI choisit un service de correspondance qui traite les documents DAML-S. L'invocation du service de correspondance par le registre UDDI est équivalente à l'invocation d'un service Web par un client. Le registre localise le service de correspondance qu'il veut invoquer en appliquant le processus de recherche standard. Une fois le service de correspondance est choisi, le registre UDDI lui envoie les URLs des descriptions retournées dans la liste des **tModels** résultant de la première recherche et l'URL de la description du service Web demandé. Le service de correspondance accomplit sa tâche de correspondance sémantique - ou syntaxique - et renvoie une liste d'URLs ordonnées par *degré de similitude* entre la description demandée et les descriptions offertes dans le registre UDDI. À son tour, le registre envoie au client la liste des **tModels** qui contiennent ces URLs dans le même ordre. En général, le premier **tModel** de la liste est celui qui correspond le plus à la description du service Web demandé. À partir de ce point, le processus de recherche devient identique au processus standard. Le client envoie une requête de type **find_service()** qui contient la clé du **tModel** choisi. Il reçoit une liste de services Web qui implémentent la description pointée par ce **tModel**. Le client choisit le service Web qu'il veut invoquer et envoie une requête de type **find_binding()** contenant la clé du service Web choisi et la

clé du `tModel`. En réponse à cette requête, il reçoit les informations nécessaires pour l'invocation de ce service (Colgrave et al., 2004).

3.4. Prise en compte de la qualité de service dans la découverte d'un service Web

Diverses propriétés de qualité de service (QoS) telles que la disponibilité, la performance et la fiabilité sont nécessaires pour faire le bon choix d'un service Web. Plusieurs initiatives ont essayé d'intégrer l'aspect qualité de service au niveau de divers standards reliés aux services Web (Tian et al., 2003), (Ludwig, 2003), (Ran, 2003), (SaikhAli et al., 2003) et (Chen et al.,2003) . Nous nous intéressons dans notre recherche aux approches qui intègrent cet aspect au niveau du processus de découverte de services Web, plus précisément au niveau des registres UDDI. Nous avons choisi parmi les approches que nous avons étudiées, deux approches qui nous semblent les plus intéressantes. Ran (Ran, 2003) propose un modèle de découverte de services Web basé sur la qualité de service qui peut co-exister avec les registres UDDI standards et qui permet de faire des recherches de services plus fines. D'autre part, ShaikhAli et al. (SaikhAli et al., 2003) proposent un modèle extensible du registre UDDI pour supporter des critères de recherche autres que ceux supportés par la version native. Dans ce qui suit, nous allons voir en détail, les deux approches décrites au-dessus.

3.4.1. Modèle de découverte de services Web basé sur la Qos

Selon Ran (Ran, 2003), le modèle de découverte de services Web basé sur les registres UDDI n'est pas fiable. Environ 48% des entrées des registres UDDI font référence à des services Web qui sont introuvables ou à des informations qui ne sont pas précises. Il ajoute que le modèle actuel limite la découverte de services Web à des critères fonctionnels. Il est possible de trouver plus d'un service Web qui offre les mêmes fonctionnalités, mais les divers services n'offrent pas nécessairement la même qualité de service. De ce fait, il est nécessaire d'incorporer la qualité de service dans le processus de découverte de services.

Ran propose un nouveau modèle qui résout ce problème. C'est un modèle qui peut co-exister avec les registres UDDI standards et qui peut servir aux applications donnant une

importance à la qualité de service (Ran, 2003). Le modèle proposé, décrit dans la Figure 3.7, est basé sur quatre rôles : i) un fournisseur qui veut publier son service Web, ii) un client qui cherche un service Web, iii) un certificateur qui vérifie la qualité de service d'un service Web avant sa publication, et iv) un nouveau registre UDDI qui diffère du registre standard par son contenu. Le nouveau registre UDDI contient une description fonctionnelle du service Web ainsi que la qualité du service qui lui est associée.

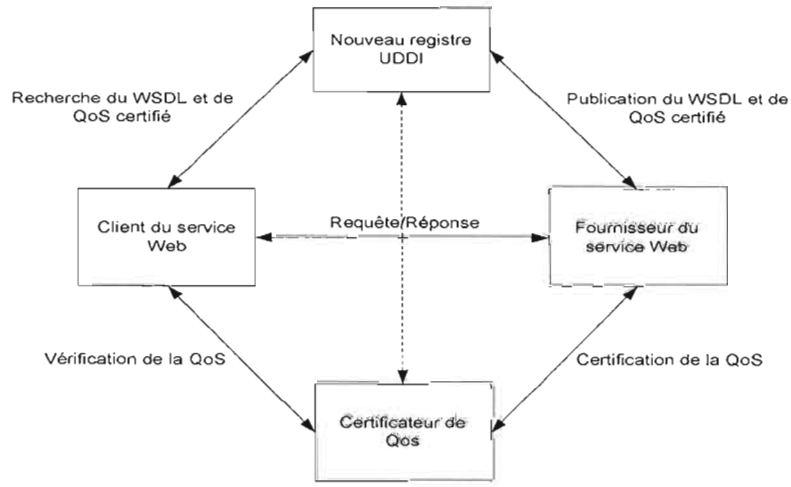

Figure 3.7. Modèle de publication et de découverte de services Web basé sur la QoS. Tirée de (Ran, 2003)

a) Publication et recherche selon le modèle de Ran

Le fournisseur commence d'abord par envoyer au certificateur de qualité les informations sur la qualité de service qu'il garantit pour le service Web qu'il offre. Le certificateur valide ces informations et les enregistre dans sa base de données en les identifiant par un *certification Id*. Par la suite, il envoie au fournisseur du service Web une certification de ces informations. Une fois que la certification est reçue, le fournisseur envoie au nouveau registre UDDI une requête de publication qui contient les informations fonctionnelles du service Web et les informations de qualité de service qui lui sont associées. Le registre UDDI communique avec le certificateur pour vérifier s'il existe une certification pour ces informations. Après vérification, le registre publie le service Web dans sa base de données.

Un client qui cherche un service Web donné, envoie une requête de recherche au nouveau registre UDDI. Cette requête se différencie d'une requête standard par l'ajout d'une partie qui décrit un besoin en terme de qualité de service. Par exemple, un client cherche un service de calculatrice qui fournit une réponse en moins de 10ms et qui est gratuit. Le nouveau registre proposé par Ran (Ran, 2003) permet de faire des recherches plus fines que sur un registre standard. Le client peut jouer sur les critères de qualité qu'il exige afin de trouver le service Web qu'il désire utiliser. Une fois que le service Web désiré est trouvé, le client vérifie les informations de qualité de service avec le certificateur en utilisant l'identificateur de certification (*certification Id*). Si le client est satisfait par la réponse, il invoque le service Web en question.

b) Extensions au registre UDDI proposées dans le modèle de Ran

Le modèle de Ran propose d'ajouter à la structure actuelle d'UDDI une entité **`QualityInformation`** qui décrit la qualité de service d'un service Web offert (Ran, 2003). Cette entité est ajoutée au niveau d'un **`BusinessService`**. Elle doit faire référence à un **`tModel`** qui lui définit une classification.

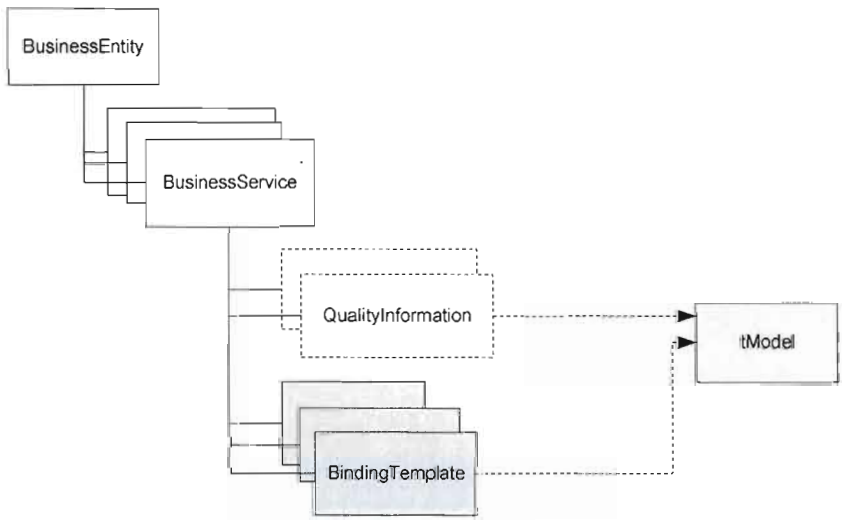

Figure 3.8. Structure de l'information selon le modèle de Ran. Tirée de (Ran, 2003)

3.4.2. UDDIe : Modèle extensible du registre UDDI

Selon ShaikhAli et al., le mécanisme de recherche standard n'est pas efficace (SaikhAli et al., 2003). Les recherches se font par nom de service ou par catégorie ce qui rend le processus découverte non efficace, il n'y a pas moyen de savoir si un service Web publié sur un registre UDDI est fonctionnel ou non. UDDIe tente de pallier aux problèmes d'UDDI. Il étend la structure standard d'UDDI afin qu'elle supporte d'autres critères de recherche. Ce modèle permet aussi de publier un service Web pour une période limitée (SaikhAli et al., 2003).

La structure d'UDDIe est définie dans un schéma XML qui étend le schéma XML d'origine proposé par dans la spécification UDDI. Quelques extensions ont été ajoutées dans la structure des enregistrements et dans les requêtes de recherche et de publication de services Web. La structure globale d'UDDIe est identique à celle d'UDDI. Les ajouts sont effectués au niveau de l'entité **`BusinessService`**. Deux types d'extensions ont été intégrés dans cette entité (Figure 3.9). Une extension **`lease`** qui permet de publier un service Web pour une période limitée et une extension **`propertyBag`** qui permet à un fournisseur de service d'ajouter un ensemble de propriétés à un service Web qu'il fournit. Une propriété **`property`** peut être un attribut de qualité de service ou une information que le fournisseur juge importante pour la découverte et l'utilisation de son service.

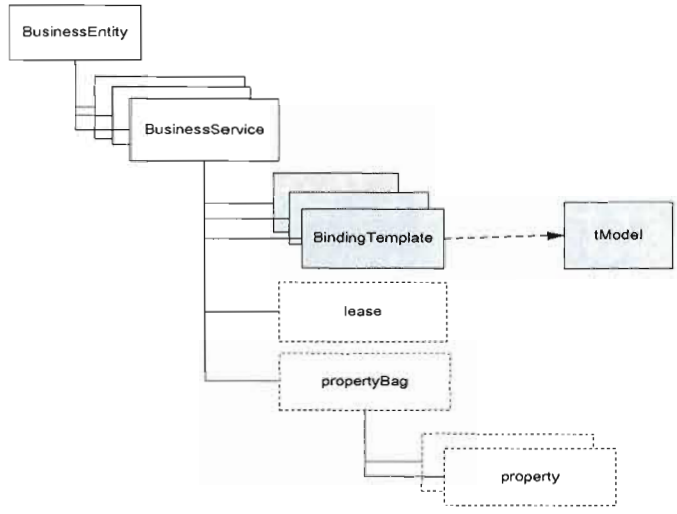

Figure 3.9. Structure d'un enregistrement dans un registre UDDIe.

L'extension `lease` (Figure 3.10) permet de préciser la période pour laquelle un service Web doit être présent sur le registre. Elle contient une date de début, une date de fin, un élément qui définit le type de location (`lease`) et un élément qui précise si la location est renouvelable ou non. Il est possible de publier un service Web pour une période infinie. Dans ce cas, le service est contrôlé par l'administrateur du registre UDDIe.

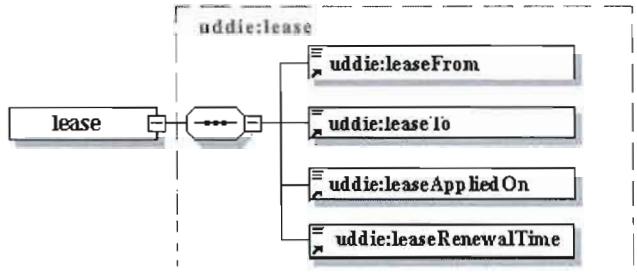

Figure 3.10. Structure de l'extension lease. Tirée de (SaikhAli et al., 2003)

L'extension `property` (Figure 3.11) permet de définir, de façon générique, une propriété liée à un service Web. Chaque propriété possède un nom, un type et une valeur.

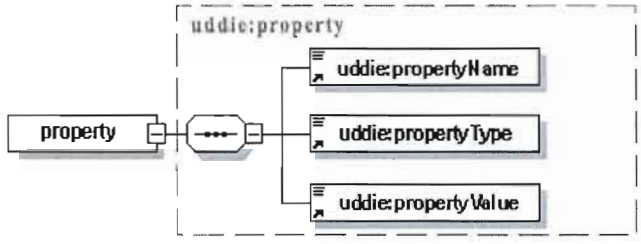

Figure 3.11. Structure de l'extension property. Tirée de (SaikhAli et al., 2003)

La Figure 3.12 montre un exemple de requête de publication sur le registre UDDIe qui enregistre un service Web en ajoutant les deux extensions décrites ci-dessus. Cette requête enregistre le service *Calculatrice* pour une période allant du 4 mai 2005 au 15 septembre 2005. Elle ne précise pas si la période de publication est renouvelable ou non. Par défaut elle n'est pas renouvelable. Cette requête enregistre aussi un ensemble de propriétés liées au service Web. A titre d'exemple, dans la Figure 3.12 la propriété qu'on veut enregistrer est nommée *CPU-Speed*, elle indique la vitesse du processeur de la machine qui héberge le service Web. On peut imaginer l'importance de ce type de propriétés dans un environnement

qui demande une rapidité d'exécution. Dans le même contexte d'idées, ShaikhAli et al. indiquent que cette technique est intéressante dans le contexte du réseau Grid (Ran, 2003).

```
<save_service xmlns="urn:uddi-org:api_v2" generic="2.0">
      <authInfo>88727f7e88727f7</authInfo>
      <businessService
           businessKey="f58ad65d-53f5-8ad6-c196-edeb4"
           serviceKey="">
           <name xml:lang="fr"> calculatrice </name>

           <lease>
                <leaseFrom> 04/05/05  10:00:00  </leaseFrom>
                <leaseTo> 15/09/05  10:00:00  </leaseTo>
           </lease>

           <propertyBag>
              <property>
                <propertyName> CPU-Speed </propertyName>
                <propertyType> number </propertyType>
                <propertyValue> 2.0 </propertyValue>
              </property>
           </propertyBag>

      </businessService>
</save_service>
```

Figure 3.12. Qualité de service dans une requête de publication

3.5. Composition dynamique de services Web

Plusieurs tentatives ont apporté de nombreux formalismes et techniques comme solutions valables pour la composition dynamique de services Web en un processus d'affaires (Zhang et al., 2003 B), (Akkiraju et al., 2001), (Srivatava et Koehler., 2003). Nous nous intéressons dans notre recherche aux approches qui intègrent une couche de composition dynamique de service Web au-dessus du registre UDDI. Nous avons choisi parmi les approches étudiées, une approche qui nous semble la plus intéressante. Zhang et al (Zhang et al., 2003 B) proposent une approche qui transforme les exigences d'un client en un processus d'affaires exécutable. Nous allons voir dans ce qui suit, les détails de cette approche.

3.5.1. Composition d'un processus d'affaire sur demande

Zhang et al. proposent un modèle de composition dynamique de services Web. Ce modèle se base sur un algorithme génétique pour transformer les besoins d'un client en un processus d'affaires exécutable (Zhang et al., 2003 B). *Web Services Outsourcing Manager (WSOM)* est un modèle qui permet la recherche, la sélection et la composition dynamiques d'un ensemble de services Web en se basant sur les exigences d'un client. Le *WSOM* traite la requête d'un client en quatre étapes. Premièrement, le modèle analyse les besoins du client. Deuxièmement, il fait une recherche de services Web demandés sur un ensemble de registres UDDI en se basant sur ces besoins. Troisièmement, il fait la sélection et la composition de services Web en se basant sur plusieurs critères. Finalement, il génère un document qui décrit le processus d'affaires équivalent aux exigences du client. Ce document représente un ensemble d'instructions compréhensibles par une machine qui sert à composer plusieurs services Web en un service Web qui réalise une tâche complexe. Dans le reste de cette section, nous allons voir en détail l'architecture proposée par Zhang et al. (Zhang et al., 2003 B). Nous présenterons, étape par étape, le processus complet proposé dans ce modèle pour générer un document descriptif d'un processus d'affaires à partir d'un besoin particulier d'un client.

a) Architecture du modèle

Le modèle *Web Service Outsourcing Manager (WSOM)* proposé par Zhang et al. se base sur deux étapes principales : une étape de recherche avancée et une étape de sélection et de composition de services Web (Zhang et al, 2003 B). La première étape consiste à faire une recherche avancée de services Web sur un ou plusieurs registres UDDI en se basant sur les exigences du client (Zhang et al, 2003 A). La deuxième étape consiste à faire une sélection et une composition de services Web en prenant comme entrée l'ensemble des services Web retourné par la première étape. La sélection et la composition dynamique de services Web se basent sur un algorithme d'optimisation qui utilise les exigences du client pour former un processus d'affaires optimisé. La Figure 3.13 montre le processus complet proposé dans ce modèle pour créer un processus d'affaires à partir des exigences d'un client. Voici étape par étape le processus au complet :

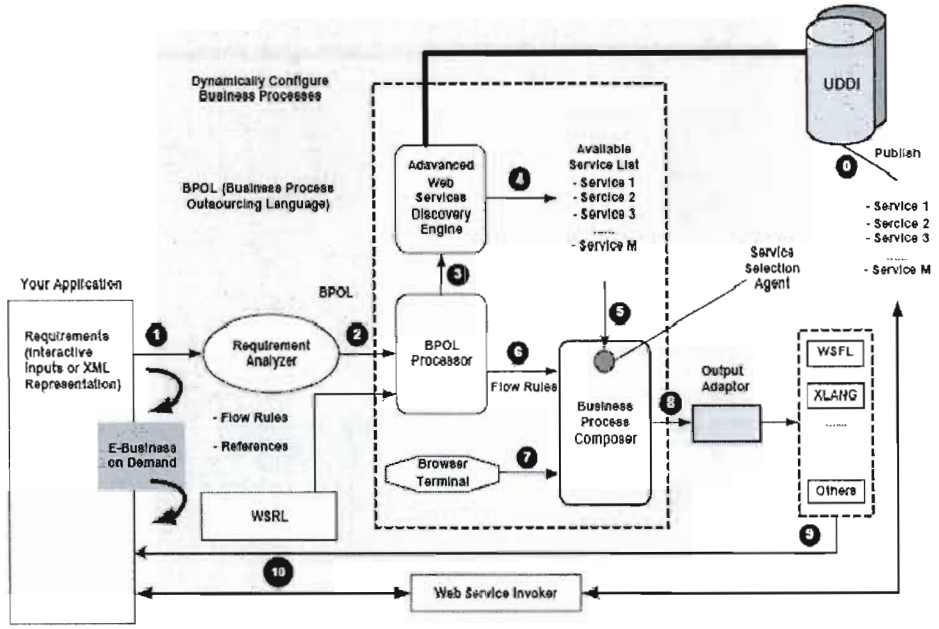

Figure 3.13. Modèle de composition dynamique de service Web. Tirée de (Zhang et al., 2003 B)

1. Le client formule ses exigences et les envoie à un analyseur d'exigences. Les exigences du client peuvent être écrites sous format XML ou via une interface interactive.

2. L'analyseur des exigences génère un document BPOL (*Business Process Outsourcing Language*) en se basant sur les exigences du client et l'envoie à un processeur BPOL. BPOL est une représentation XML qui décrit les règles de flux de service et les exigences d'affaires d'un client. Les exigences du client incluent principalement les préférences du client, les règles d'affaires que le client a définies, et les liens entre les services Web (Zhang et al, 2003 A). Les règles de flux de service représentent le modèle de composition prévu par le client. Les préférences du client incluent tout type d'information représentant ses choix. Dans cette partie, le client peut préciser les registres UDDI sur lesquels il préfère faire sa recherche. Les préférences du client peuvent aussi contenir des informations sur les qualités de service exigées par

ce dernier. Les liens entre services Web décrivent la relation entre ces derniers à plusieurs niveaux. Une relation peut être entre un service Web et un autre ou entre une opération d'un service et un service Web. Les liens entre services Web sont décrits dans un document de type WSRL (*Web Services Relationship language*) pour ensuite y faire référence dans le document BPOL. La Figure 3.14 montre un exemple d'un document BPOL. Un document de ce type est utilisé pour la recherche avancée de services Web et pour la sélection et la composition de ces derniers.

3. Le processeur BPOL analyse le document BPOL reçu et génère un document USML (*UDDI Search Markup Language*) en se basant sur ce dernier. USML est un langage de scripts basé sur XML qui permet de faire des recherches multiples sur des registres UDDI. Le document USML généré est envoyé à un engin de recherche avancée de services Web (Zhang et al, 2003 A).

4. L'engin de recherche avancée analyse le document USML qu'il reçoit et fait une recherche sur les registres UDDI spécifiés. Les critères de recherche sont spécifiés dans le document USML. Le résultat est une liste de services Web.

5. La liste des services Web obtenue à l'étape 4 est passée à un « compositeur de processus d'affaires ».

6. Le processeur BPOL extrait les règles de flux de service exigées par le client à partir du document BPOL et les envoie au compositeur de processus d'affaires. À ce niveau le compositeur détient l'information nécessaire pour sélectionner et composer un processus d'affaires optimal à partir d'une liste de services Web. La sélection de services est réalisée en appliquant un algorithme génétique.

7. Le client peut confirmer le choix et la composition de services proposée par le compositeur de processus d'affaires via une interface Web ou une interface graphique.

8. Le résultat de la composition est un processus d'affaires qui décrit un ensemble d'interactions entre les services Web sélectionnés. Un adaptateur de sortie permet de formater ce processus sous un langage qui permet son exécution (e.g BPEL ou WSFL).

9. Le document qui décrit le processus d'affaires est retourné au client comme résultat à ces exigences.

10. Le client peut exécuter le processus résultant en envoyant le document qui le décrit à un engin qui permet l'exécution de processus d'affaires. Par exemple, si le document qui décrit le processus d'affaires généré est écrit en BPEL alors on envoie ce document à un engin d'exécution de processus d'affaires qui comprend le langage BPEL (BPEL4WS,2002).

```
<!DOCTYPE BPOL SYSTEM "BPOL.dtd" >
<BPOL>
<FlowRule>
<Parallel>
<ParallelNumber>0</ParallelNumber>
<ParallelTask>Match</ParallelTask>
<ParallelTask>Food</ParallelTask>
</Parallel>
<Sequential>
<SequentialNumber>5</SequentialNumber>
<SequentialTask>Start</SequentialTask>
<SequentialTask>News Collection</SequentialTask>
<SequentialTask>Food</SequentialTask>
<SequentialTask>End</SequentialTask>
</Sequential>
......
</FlowRule>
</BPOL>
```

Figure 3.14. Exemple d'un document BPOL

b) Recherche avancée de services Web

Rappelons que les services Web candidats à la composition sont recherchés grâce à un engin de recherche avancée *AUSE* (*Advanced UDDI Search Engine*). Dans cette section, nous décrivons le fonctionnement de cet engin. La Figure 3.15 montre l'architecture du modèle permettant ce type de recherche (Zhang et al., 2003 A).

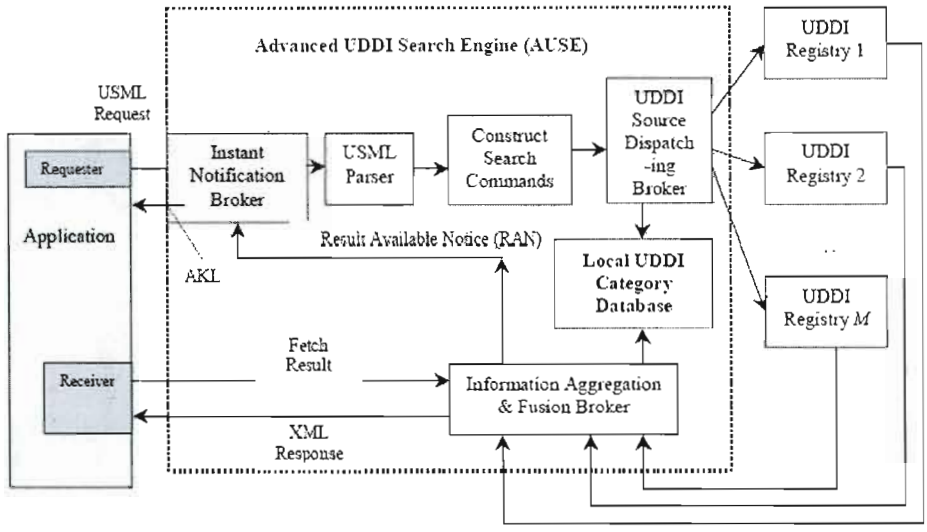

Figure 3.15. Modèle de recherche avancée. Tirée de (Zhang et al., 2003 A)

L'engin reçoit en entrée une requête sous format USML. Rappelons que USML est un langage de script qui permet de spécifier des requêtes pour faire des recherches multiples sur plusieurs registres UDDI (Zhang et al., 2003 A) . Le composant 'instant notification broker' représente l'interface entre le client et le modèle. Une requête USML reçue par ce composant est envoyée à un parseur qui va l'analyser. Un ensemble de requêtes UDDI standard sont ensuite construites et envoyées à un 'UDDI Source Dispatching Broker' qui va les acheminer automatiquement aux registres UDDI spécifiés par le client. Notons qu'il est possible d'envoyer les requêtes à un registre UDDI local intégré dans le modèle. Les résultats retournés par les registres UDDI sont combinés au niveau d'un 'Fusion Broker' selon un type d'agrégation demandé par le client. Une agrégation peut être une UNION ou une INTERSECTION entre les résultats retournés par les registres UDDI. Une fois l'agrégation effectuée, le 'Fusion Broker' envoie au 'Instant Notification Broker' un avis pour l'informer qu'un résultat est disponible. Le composant 'Instant Notification Broker ' envoie à son tour un avis au client pour l'informer qu'il peut aller chercher le résultat au niveau du 'Fusion Broker'. Le résultat final est retourné au client sous format USML. Les Figure 3.16 et Figure 3.17 montrent respectivement un exemple d'entrée (requête USML) et un exemple de sortie (réponse USML) du modèle de recherche avancée.

```
<?xml version="1.0"?>
<!DOCTYPE Search SYSTEM "UDDISearch.dtd">
<Search>
  <ProcessId>9999</ProcessId>
  <Query>
    <Source>Private UDDI</Source>
    <SourceURL>http://wsbi10/services/uddi/inquiryAPI</SourceURL>
    <BusinessName>Air France </BusinessName>
    <FindBy>Business</FindBy>
  </Query>
  <Query>
    <Source>Public UDDI</Source>
    <SourceURL>http://wsbi5/services/uddi/servlet/uddi</SourceURL>
    <BusinessName> Hertz </BusinessName>
    <FindBy>Business</FindBy>
  </Query>
  <AggOperator>OR</AggOperator>
</Search>
```

Figure 3.16. Exemple de requête USML

```
<?xml version="1.0"?>
<USMLResponse>
    <Business>
        <BusinessName> Air France </BusinessName>
        <BusinessKey>
                339CFD70-E84D-11D5-B61C-970107B90C83
        </BusinessKey>
        <Description> Compagnie aérienne </Description>
        <URL>
           http://wsbi10/services/uddi/uddiget?businessKey=
           339CFD70-E84D-11D5-B61C-970107B90C83
        </URL>
       <Operator>wsbi10/services/uddi</Operator>
    </Business>
    <Business>
        <BusinessName> Hertz </BusinessName>
        <BusinessKey>
                8EF91680-EDDA-11D5-ADF1-850C07A41C41
        </BusinessKey>
        <Description>Compagnie de location de voiture</Description>
        <URL>
            http://wsbi5:80uddiget?businessKey=
            8EF91680-EDDA-11D5-ADF1-850C07A41C41
        </URL>
        <Operator>wsbi5:80/services/uddi</Operator>
    </Business>
</USMLResponse>
```

Figure 3.17. Exemple de réponse USML

c) Sélection et composition dynamique de services Web

Rappelons que le but principal de cette approche est de configurer un processus d'affaires réel à partir de la composition d'un ensemble de services abstraits proposé par le client (Zhang et al., 2003 B). Le mécanisme de découverte et de sélection offert par cette approche se base sur les exigences du client pour construire un processus d'affaires concret. L'approche se base sur la notion de service Web abstrait, qui représente un ensemble de services Web concrets offerts par plusieurs fournisseurs et qui réalisent une même tache (Figure 3.18). Par exemple, le service abstrait 'Achat billet avion' comprend l'ensemble des services Web qui permettent de faire des achats de billets d'avion. Des services Web offerts par 'Air France' et par 'Air Canada' sont des instances de ce service abstrait.

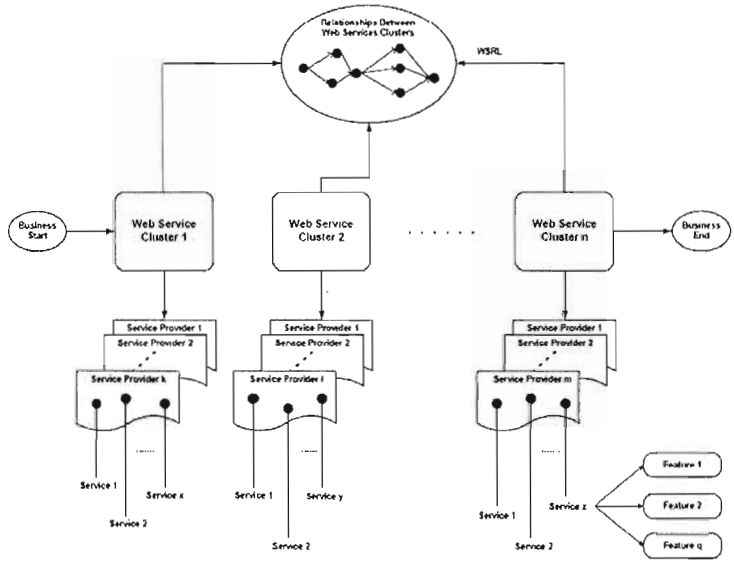

Figure 3.18. Composition dynamique de services Web. Tirée de (Zhang et al., 2003 B).

Selon l'approche, un seul service Web concret est sélectionné par service abstrait. Par exemple si un processus d'affaires comporte trois services abstraits alors trois services Web concrets seront sélectionnés.

Rappelons que le processus d'affaires tel qu'exigé par un client peut consister en plusieurs services abstraits qu'on a assujettis à des contraintes. La recherche peut identifier

plusieurs services concrets candidats par service abstrait. Zhang et al. proposent de trouver la combinaison optimale en utilisant un algorithme génétique (Zhang et al, 1995). Dans cet algorithme génétique, un chromosome est équivalent à un processus d'affaires. Chaque gène présent dans un chromosome est équivalent à un service Web présent dans un processus d'affaires. Un gène possède deux valeurs possibles, 0 pour dire que le gène n'est pas présent dans le chromosome et 1 pour dire que le gène est présent dans ce dernier. La combinaison de l'ensemble des valeurs des gènes forme le chromosome. Dans cette approche, Zhang et al. modifient la représentation traditionnelle d'un chromosome pour définir la composition de services en un processus d'affaires. Selon eux, un service Web peut être offert par plusieurs fournisseurs et de ce fait, on ne sait pas, a priori, quels sont les services Web déjà sélectionnés. Donc, un gène d'un chromosome est plutôt équivalent à un service Web abstrait d'un processus d'affaires. Selon l'approche, un processus d'affaires est équivalent à un chromosome représenté comme suit :

$$Chromosome=[S_{11}\ S_{12}\ \ldots S_{1i}\ |\ S_{21}\ S_{22}\ \ldots S_{2j}|\ \ldots\ \ S_{n1}\ S_{n2}\ \ldots S_{nk}|]$$

Où n représente le nombre de services abstraits et i,j,k représente le nombre de services Web candidats pour chaque service abstrait. S_{nk} représente un service Web concret dont la valeur est égale à 1 pour dire que le service est sélectionné ou à 0 pour dire que le service n'est pas sélectionné.

Rappelons qu'un processus d'affaires est une composition concrète d'un ensemble de services Web sélectionnés à partir des services abstraits qui définissent ce processus. Selon l'approche, au plus un service Web est sélectionné par service abstrait. Donc de l'ensemble (S_{11}, S_{12}, ..., S_{1i}) il y'aura au plus un seul service sélectionné. Il en sera de même pour les autres ensembles.

La composition d'un processus d'affaires doit respecter les relations entre les services Web définis au niveau du document BPOL et sur la richesse fonctionnelle offerte par chacun de ces services. L'algorithme génétique utilisé pour cette composition cherche à optimiser deux critères : le temps - combien de temps ça va prendre? - et le coût - combien ça va coûter - . Ces deux critères sont les premiers qui intéressent le client (Zhang et al, 2003 B). En se

basant sur ces deux critères, l'algorithme choisit la composition de services qui coûte le moins en terme de temps et de coût. L'algorithme fait une comparaison entre toutes les compositions possibles. La solution retournée est une séquence binaire qui représente la composition choisie.

3.6. Discussion

Plusieurs approches ont essayé d'étendre les registres UDDI afin d'apporter plus de souplesse. Dans notre discussion nous avons classé les approches décrites par type de solution apportée. Un premier type de solution consiste à utiliser le même API de UDDI, mais d'étendre le comportement d'exécution lorsqu'une requête inclut une nouvelle information (Colgrave et al., 2004). Ce type de solution a comme avantage la transparence vis-à-vis des clients. L'interaction des clients avec ce type de registre se fait de la même façon qu'avec un registre UDDI standard. Cependant, ce type de solution possède deux problèmes majeurs. Premièrement, le type d'extensions est limité par les signatures (paramètres) des fonctions existantes. Deuxièmement, la réalisation de cette solution nécessite d'implémenter à nouveau un registre UDDI.

Un deuxième type solution consiste à implanter un registre UDDI avec une API étendue afin de supporter de nouvelle fonctionnalité (Srinivasan et al., 2004) et (SaikhAli et al., 2003). L'avantage de ce type de solution c'est que le client peut utiliser soit la version standard, soit la version étendue. L'inconvénient majeur de cette solution est qu'elle nécessite l'implantation à nouveau d'un registre UDDI. Par exemple, la solution apportée par Srinivasan et al. nécessite l'ajout d'un port d'entrée au registre UDDI[1] pour permettre aux clients de soumettre et exécuter des requêtes de recherche de services Web qui tiennent compte de descriptions sémantiques de type OWL-S (Srinivasan et al., 2004).

---

[1] Rappelons qu'un registre UDDI est, *lui-même*, un service Web.

Un troisième type de solution consiste à utiliser un intermédiaire entre les registres UDDI standard et les clients qui les utilisent. Les clients qui veulent utiliser une nouvelle fonctionnalité communiquent avec une application intermédiaire qui fait un traitement local pour ensuite déléguer une partie de son travail à un ou plusieurs registre UDDI standard. Par exemple, Zhang et al. (Zhang et al., 2003 B) proposent une solution qui permet à un client d'envoyer des requêtes de nature complexe. Ce type de solution est le moins intrusif, et garantit une meilleure compatibilité avec les registres UDDI existants. De plus, ce type de solution n'exige pas de changer ni le contenu ni l'implantation des registres UDDI. Par contre, un client doit parler à différents APIs spécifiques à chacune des extensions qu'il veut utiliser (QoS, Composition dynamique, etc.). Il doit interagir avec plusieurs intermédiaires distants et doit gérer les interactions localement. Ceci rend la tâche du client plus lourde.

Notre objectif est de proposer un mécanisme d'extension générique et non intrusif qui permette d'ajouter plusieurs extensions à un registre UDDI. L'idée est d'introduire un registre intermédiaire entre les clients et les registres UDDI qui supporte l'ajout de nouveaux types de requêtes non définies dans la version standard du registre UDDI. Chaque nouvel ajout représente une nouvelle extension au registre UDDI.

CHAPITRE IV

JRegistre : un registre UDDI extensible

4.1. Introduction

Les implémentations des registres UDDI existantes permettent d'effectuer un nombre de requêtes simples sur ces derniers afin de récupérer de l'information sur le domaine d'affaire d'une compagnie ou sur les services offerts par celle-ci. Pour publier ou récupérer de l'information complexe, un utilisateur se doit d'effectuer plusieurs requêtes simples sur un registre UDDI. Ceci est dû au fait que UDDI propose seulement des requêtes de base permettant d'effectuer une tâche à la fois sur le registre et ne permet pas d'intégrer des requêtes complexes. Pour pallier à cette limitation, nous proposons une approche baptisée sous le nom de « JRegistre » afin d'apporter une souplesse à l'utilisation des registres UDDI (Mili et al.,2005). Cette solution consiste en une extension aux registres UDDI qui supporte l'ajout dynamique de requêtes spécifiques à l'exécution (run-time). Grosso modo, notre solution consiste à fournir un registre intermédiaire qui agit comme une interface entre les registres UDDI standard et les clients désirant faire des requêtes complexes (Figure 4.1).

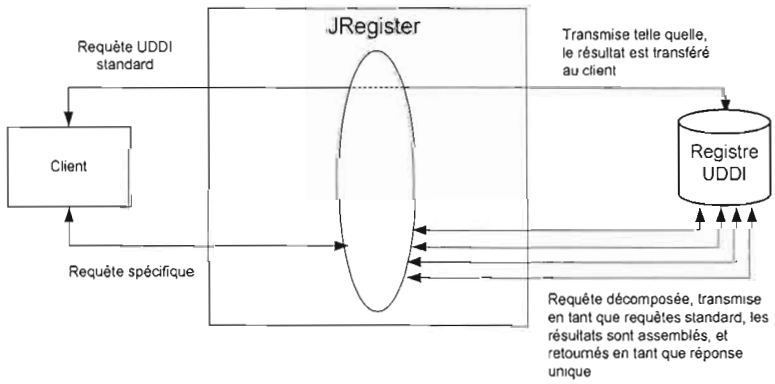

Figure 4.1. JRegistre : un registre intermédiaire

4.2. Architecture globale

La Figure 4.2 montre l'architecture globale de « JRegistre ». Elle se divise en deux parties. La première partie est composée d'une « Interface de publication de requêtes complexes » et d'un « Générateur de requêtes complexes », qui servent pour la génération de nouvelles requêtes. La deuxième partie est composée d'un « Proxy JRegistre », d'un « Moteur de requêtes » et d'un « Registre UDDI » qui servent pour l'invocation des requêtes complexes générées et des requêtes UDDI standards. Ainsi, un client, qui veut générer une nouvelle requête, utilise l'interface de publication fournie par ce modèle. Il envoie la description de la requête, la description de la réponse ainsi que la logique fonctionnelle de la requête qu'il veut générer, à un « Générateur de requêtes complexes ». Ce générateur analyse les données reçues, génère la requête et l'ajoute à l'ensemble des requêtes complexes gérées par le moteur de requêtes. Il est important de noter que l'ajout se fait automatiquement pour permettre au système d'être à jour, sans être obligé de le réinitialiser chaque fois qu'une nouvelle requête est ajoutée. De ce fait, une nouvelle requête est fonctionnelle à partir du moment de son intégration.

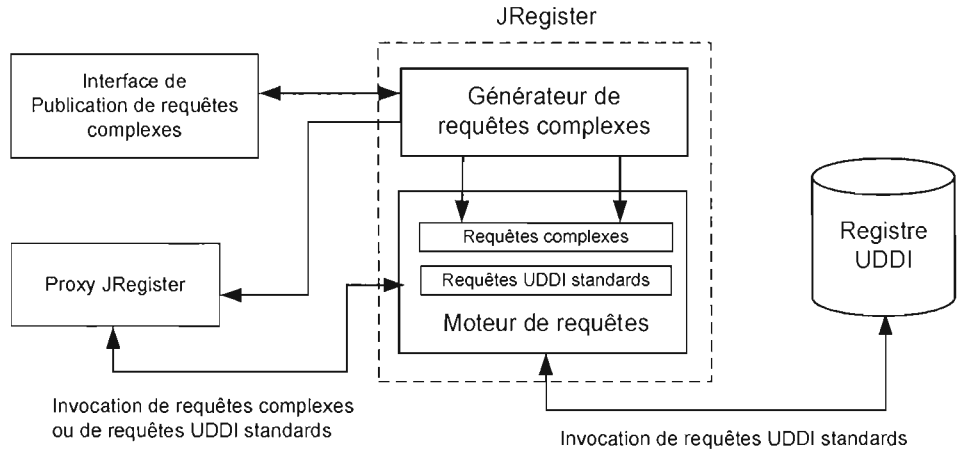

Figure 4.2. Architecture globale de JRegistre

Le composant « Proxy JRegistre » sert comme interface d'invocation pour l'ensemble des requêtes offertes par « JRegistre » (requêtes UDDI standard et requêtes complexes). Ce composant est également mis à jour chaque fois qu'une nouvelle requête est ajoutée. Deux

alternatives sont alors possibles pour la mise à jour de ce composant. La première alternative consiste à mettre à jour automatiquement le « Proxy JRegistre » offert par le système. Ainsi, un client qui veut avoir un « Proxy JRegistre » à jour, doit obligatoirement le télécharger à nouveau. La deuxième alternative ajoute à la première une étape qui avise les clients qui utilisent déjà le registre des éventuelles nouvelles requêtes ajoutées au système. Il leur signale qu'ils doivent mettre à jour le composant « Proxy JRegistre » s'il désirent avoir accès aux requêtes ajoutées.

Le client peut invoquer les requêtes standard UDDI ou les des requêtes complexes via l'interface d'invocation « Proxy JRegistre ». Toutes les requêtes sont envoyées au « Moteur de requêtes » au niveau de JRegistre. Si la requête est de type UDDI standard alors elle est transmise telle quelle à un registre UDDI standard. Le résultat sera transféré tel qu'il est au client. Dans le cas où la requête est de type complexe, elle est décomposée en un ensemble de requêtes standard. Ces requêtes sont transmises au registre UDDI pour traitement. les réponses à ces requêtes seront assemblées et par la suite retournées au client en tant que réponse unique (Figure 4.1).

Pour des raisons de réutilisation et de portabilité, nous avons choisi pour le registre introduit dans notre approche, de nous inspirer de la même structure et de la même logique de développement définies dans jUDDI. Notons que jUDDI est une implémentation Java à code source ouvert de la spécification UDDI. Par ailleurs, « JRegistre » respecte la même architecture, que celle utilisée dans jUDDI et réutilise autant que possible les composantes de ce dernier. Nous allons introduire dans la section suivante une description détaillée de la structure et du mode de fonctionnement du registre jUDDI.

4.3. *jUDDI : Une implantation Java du registre UDDI*

C'est une implémentation « open source » qui représente le serveur UDDI officiel de la fondation Apache. jUDDI est une implémentation de l'ensemble des requêtes de publication et de recherche standard définies dans la version 2.0 de la spécification UDDI. La Figure 4.3 montre l'architecture globale de cette implémentation.

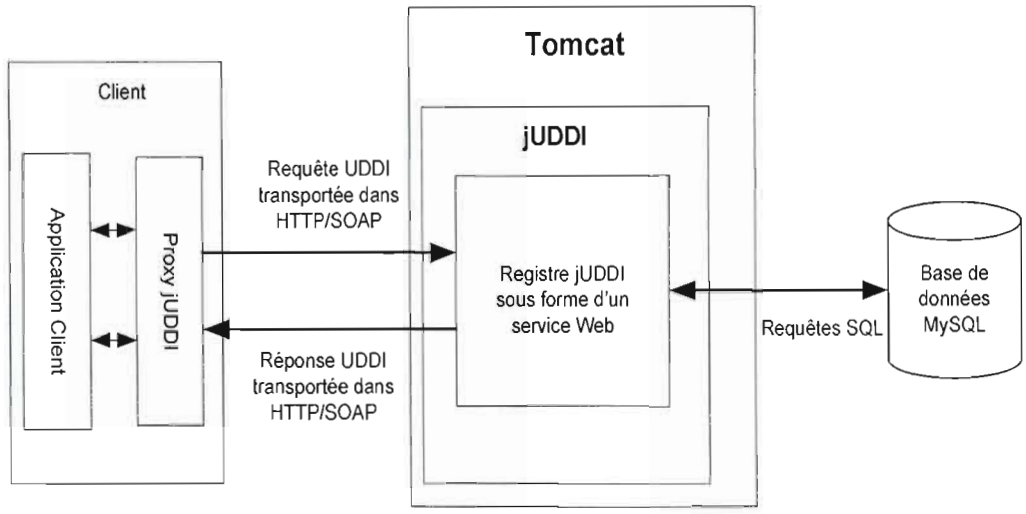

Figure 4.3. Architecture globale du registre jUDDI

jUDDI utilise une architecture orientée service Web. Il offre un « Proxy jUDDI » qui sert comme interface d'invocation pour les requêtes UDDI standard. L'envoi et la réception de requêtes et de réponse UDDI entre le proxy et le registre jUDDI se font par envoi de messages SOAP via HTTP. Une requête UDDI reçue par le registre jUDDI est convertie en un ensemble de requêtes SQL, envoyées ensuite à une base de données de type MySQL. Le résultat est construit au niveau du registre jUDDI puis est retourné au proxy. Notons qu'il est possible d'utiliser d'autres bases de données. Cela dit, Apache recommande d'utiliser les bases de données MySQL. Nous nous intéressons dans cette architecture aux interactions entre le « Proxy jUDDI » et le registre jUDDI. Autrement dit, au processus d'échange de messages XML représentant les requêtes et les réponses entre le proxy et le registre, et aux éléments qui interviennent dans ce processus. Rappelons que le processus d'échange entre le « Proxy JRegistre » et « JRegistre » définis dans notre approche se base sur celui défini par jUDDI. Il est donc important de comprendre les étapes déterminantes de ce processus. Nous allons décrire, dans ce qui suit, le processus complet de requête/réponse entre le « proxy jUDDI » et le « registre jUDDI ». Les Figure 4.4 et Figure 4.5 décrivent respectivement les étapes d'envoi d'une requête et de réception d'une réponse au niveau du proxy, et les étapes de réception d'une requête et d'envoi d'une réponse au niveau du registre.

Processus d'envoi d'une requête et de réception d'une réponse au niveau du Proxy

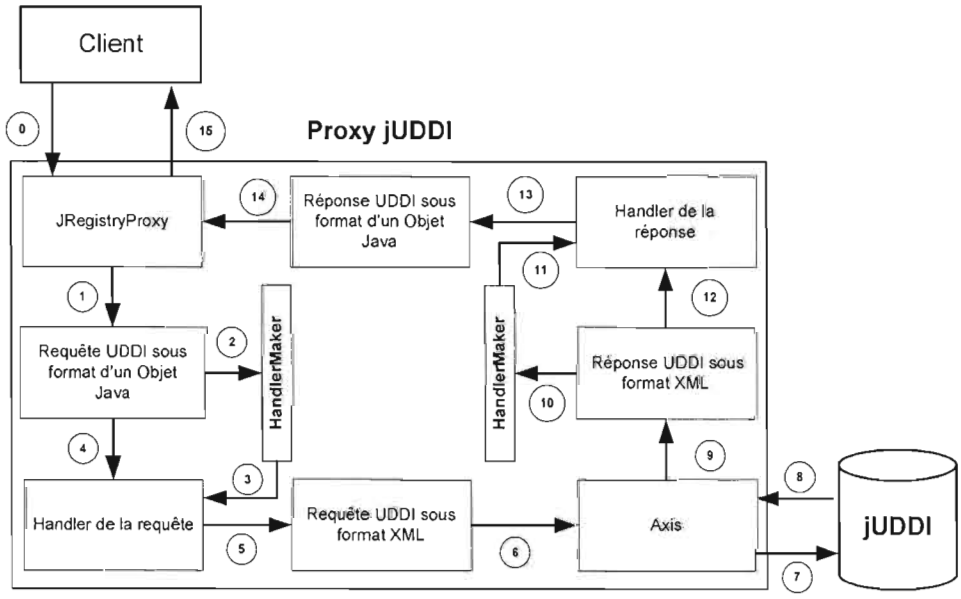

Figure 4.4. Processus d'envoi et de réception au niveau du Proxy

L'élément *JregistryProxy* (Figure 4.4) représente l'interface client permettant d'invoquer les requêtes UDDI implantées par jUDDI. Cette interface définit l'ensemble des requêtes UDDI implantées par un ensemble de méthodes Java. Chaque méthode correspond à une requête et retourne une réponse. Pour rappel, jUDDI implante en Java, la version 2.0 de la spécification UDDI, et il respecte, dans son implantation, le document WSDL offert par cette spécification.

Pour envoyer une requête au registre jUDDI (Figure 4.4), le client choisit la méthode qui la représente au niveau du *JregistryProxy* (0). Selon les informations contenues dans cette méthode, un objet Java représentant la requête est construit (1). Il est à noter que selon la spécification UDDI, les messages échangés entre un client et un registre UDDI doivent être sous format XML. De ce fait, jUDDI offre un mécanisme de transformation appelé *Handler* qui transforme une requête –ou une réponse– UDDI sous forme d'un objet Java en une requête –une réponse– UDDI sous format XML et vice versa. Chaque requête et chaque réponse possèdent leur propre *Handler* qui permet la transformation dans les deux sens. Le choix du *handler* est assuré par l'intermédiaire d'un *HandlerMaker*. Selon le nom de la

requête sous format Java (2), le *HandlerMaker* choisit le *Handler* spécifique à cette requête (3). Une action de *Marshaling* est appliquée sur l'objet Java représentant la requête (4) pour le transformer en une requête XML (5). Cette requête est encapsulée dans une enveloppe SOAP (6) et ensuite envoyée au registre jUDDI via le protocole HTTP (7).

Une réponse reçue du registre jUDDI est traitée de la façon inverse (8). Une enveloppe SOAP est extraite à partir de la réponse HTTP reçue, ensuite de quoi une réponse UDDI sous format XML est extraite de l'enveloppe SOAP (9). À partir du nom de la réponse XML reçue (10), le *HandlerMaker* choisit le *Handler* spécifique à cette réponse (11). Une action de *Unmarshaling* est appliquée à la réponse XML (12) pour la transformer en un objet Java la représentant (13). Cet objet est retourné comme réponse à la méthode représentant la requête invoquée (14). Enfin, le client peut récupérer les informations contenues dans la réponse (15).

Processus de réception d'une requête et d'envoi d'une réponse au niveau du registre

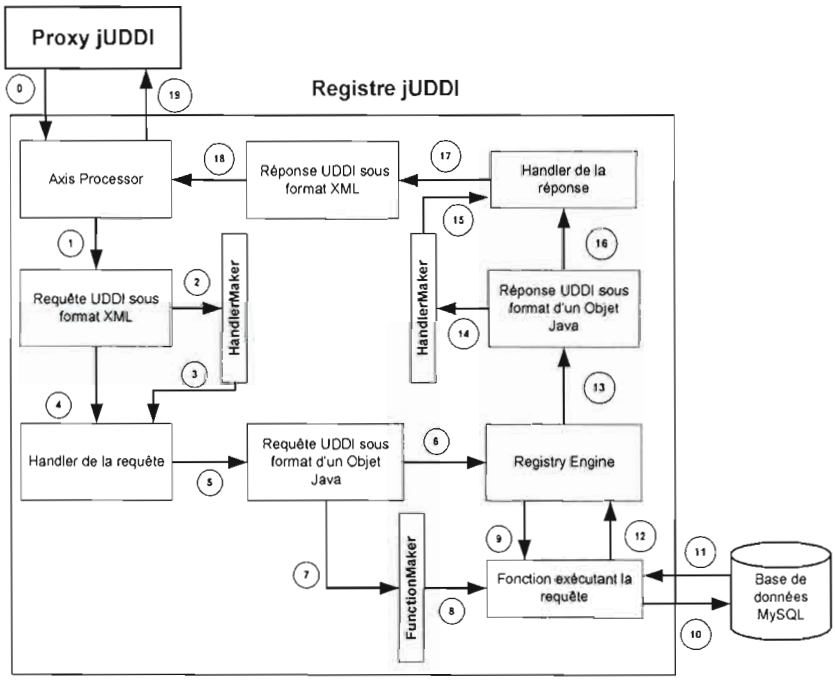

Figure 4.5. Processus de réception et d'envoi au niveau du registre

La Figure 4.5 montre le processus de réception d'une requête et d'envoi d'une réponse au niveau du registre jUDDI. Les requêtes sont reçues, au niveau du registre, sous forme de requêtes HTTP (0). Après réception, le registre procède à la transformation inverse de la requête. À l'issue de cette transformation, on obtient une requête sous forme d'un objet Java. Plus en détail, une enveloppe SOAP est extraite à partir de la requête HTTP reçue. Celle-ci est, à son tour, traitée pour en extraire une requête UDDI sous format XML (1). À partir du nom de la requête XML reçue (2), le *HandlerMaker* choisit le *Handler* spécifique à cette requête (3). Une action de *Unmarshaling* est appliquée sur la requête XML (4) pour la transformer en un objet Java qui la représente (5). La requête sous format Java est ensuite envoyée à un *Registry Engine* qui s'occupe de son exécution (6). Selon le nom de la requête (7), le composant *FunctionMaker* choisit la fonction qui définit la logique fonctionnelle derrière la requête (8). Par la suite, le *Registry Engine* exécute la fonction choisie (9). Notons que la logique fonctionnelle de la requête est généralement composée d'un ensemble de requêtes SQL exécutées sur une base de données MySQL (10).

À partir des résultats des requêtes SQL exécutées sur la base de données (11), une réponse sous forme d'un objet Java est construite au niveau de la fonction (12) (13). Selon le type de réponse (14), le *HandlerMaker* choisit le *Handler* qui lui correspond (15). Une action de *Marshaling* est appliquée sur l'objet Java représentant la réponse (16) pour le transformer en une réponse XML (17). Cette réponse est encapsulée dans une enveloppe SOAP (18) et envoyée au proxy via le protocole HTTP (19).

La figure ci-dessous (Figure 4.6) résume le processus complet de transformation d'une requête du moment de sa création au niveau du proxy jUDDI jusqu'au moment de son exécution sur le registre jUDDI et celui d'une réponse du moment de sa création jusqu'au moment de sa réception par le client.

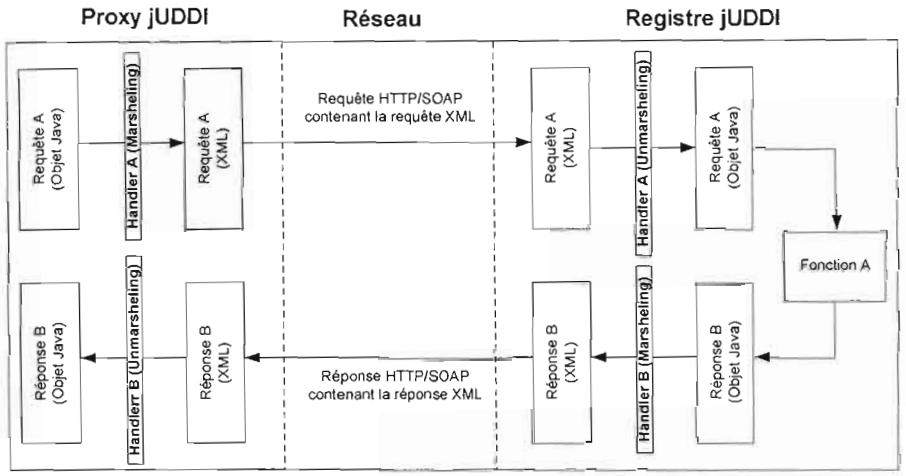

Figure 4.6. Processus de transformation d'une requête et de sa réponse

Nous allons voir dans la suite de cette section une description détaillée des éléments qui interviennent dans le processus d'envoi et de réception d'une requête/réponse sur un registre jUDDI (Figure 4.4 et Figure 4.5).

4.3.1. Interface d'invocation des requêtes UDDI au niveau du proxy jUDDI

Le proxy jUDDI possède une interface **`IRegistry`** qui définit l'ensemble des requêtes UDDI qui peuvent être invoquées. Chaque requête est représentée par une méthode qui prend en paramètre le contenu de la requête et qui retourne une réponse à celle-ci. Par exemple, la requête **`find_service()`** est définie au niveau de l'interface **`IRegistry`** par la méthode **`findService()`** décrite dans la figure ci-après (Figure 4.7).

```
public interface IRegisry {
...

ServiceList findService(String businessKey,
                        Vector nameVector,
                        CategoryBag categoryBag,
                        TModelBag tModelBag,
                        FindQualifiers findQualifiers,int maxRows)
                        throws RegistryException;
...
}
```

Figure 4.7. Méthode représentant la requête find_service() au niveau de jUDDI

La classe **AbstractRegistry** contient l'implémentation de toutes les méthodes définies dans l'interface **IRegistry**. Ces méthodes permettent de construire l'objet représentant la requête, pour ensuite faire appel à la méthode **execute()** définie dans la classe **RegistryProxy**. Cette méthode permet de se connecter au registre, d'envoyer la requête à ce dernier et d'en recevoir la réponse. La Figure 4.18 montre la hiérarchie de classes définies ci-dessus.

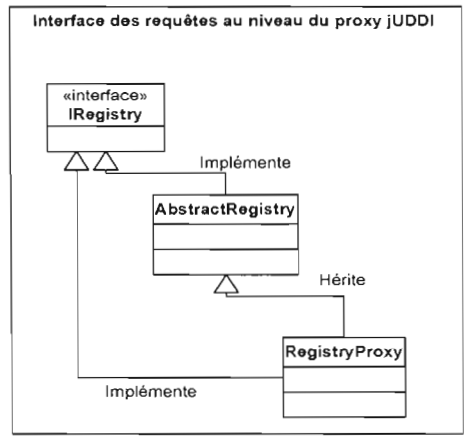

Figure 4.8. Hiérarchie des classes permettant d'invoquer les requêtes UDDI

4.3.2. Structure d'une requête et de sa réponse au niveau du registre jUDDI

Telles qu'il a été mentionné dans la section précédente, les requêtes et les réponses UDDI standard sont représentées au niveau du registre jUDDI par des classes Java. Chaque requête - réponse - correspond à une classe Java qui porte le même nom que celle-ci. Les attributs de cette classe sont de types simples (String, int, float,...) ou de types complexes. Cela dépend de la structure de la requête - réponse - sous format XSD. Rappelons que la spécification UDDI définit les requêtes et les réponses UDDI en utilisant les schémas XML (XSD). Les requêtes et les réponses Java implémentent une interface **RegistryObject**. Les Figure 4.9 et Figure 4.10 montrent respectivement, le schéma XML de la requête **find_service()** défini au niveau de la spécification UDDI, et un extrait de code de la classe Java représentant cette requête au niveau du registre jUDDI.

```
<xsd:complexType name="find_service">

    <xsd:sequence>
      <xsd:element ref="uddi:findQualifiers" minOccurs="0"/>
      <xsd:element ref="uddi:name" maxOccurs="unbounded"/>
      <xsd:element ref="uddi:categoryBag" minOccurs="0"/>
      <xsd:element ref="uddi:tModelBag" minOccurs="0"/>
    </xsd:sequence>
    <xsd:attribute name="generic" type="string" use="required"/>
    <xsd:attribute name="maxRows" type="int" use="optional"/>
    <xsd:attribute name="businessKey" type="uddi:businessKey"
                                              use="optional"/>
</xsd:complexType>
```

Figure 4.9. La requête find_service() sous format XSD

```
public class FindService implements RegistryObject,Inquiry
{
  // attribut
  String generic;
  String businessKey;
  Vector nameVector;
  CategoryBag categoryBag;
  TModelBag tModelBag;
  FindQualifiers findQualifiers;
  int maxRows;

  // Constructeurs
  public FindService()
  {…}
  // les getters et les setters
  }
```

Figure 4.10. Classe Java représentant la requête find_service()

4.3.3. Structure des handlers

Les *handlers* sont les éléments responsables de la transformation d'une requête UDDI sous format XML en un objet Java et vice versa. Chaque requête - réponse - possède son propre *handler*, représenté par une classe Java, qui assure la transformation dans les deux sens. Toutes les classes héritent de la classe **`AbstractHandler`** qui implémente elle-même une interface **`IHandler`** qui définit deux fonctions. Une première fonction de « Marshaling » qui prend un élément DOM représentant la requête - réponse - et qui copie son contenu dans une instance de la classe Java représentant cette requête - réponse -. Une deuxième fonction de « Unmarshaling » qui copie le contenu d'un objet Java représentant une requête dans un

élément DOM. La Figure 4.11 montre la hiérarchie générale des *handlers* au niveau de jUDDI.

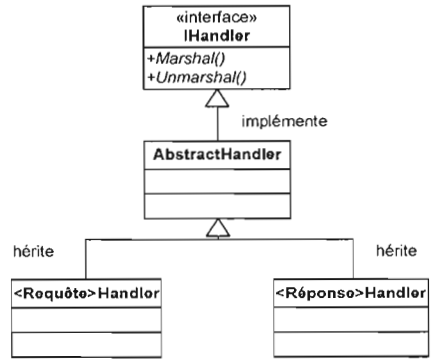

Figure 4.11. Hiérarchie des handlers au niveau de jUDDI

Prenons l'exemple de la Figure 4.12. La classe **`FindServiceHandler`** représente le *handler* responsable de la transformation de la requête **`find_Service()`** dans les deux sens.

```
public class FindServiceHandler extends AbstractHandler
{
  …
  public RegistryObject unmarshal(Element element)
  {
    FindService obj = new FindService();
    …
    return obj;
  }
  public void marshal(RegistryObject object,Element parent)
  {…}
}
```

Figure 4.12. Class java représentant le handler de la requete find_service()

Utilisation de DOM comme élément intermédiaire entre XML et Java

DOM, pour *Document Object Model* (DOM-W3C, 2005), est une spécification sous forme d'un langage neutre qui permet de construire une représentation d'un document XML sous forme d'arbre organisé hiérarchiquement en mémoire. DOM a pour fonction d'étendre les possibilités de manipulation des documents XML. Les développeurs de jUDDI ont utilisé le parseur DOM nommé « Xerces » pour permettre le chargement d'une requête - réponse - sous format XML dans un arbre DOM en mémoire. Notons que les nœuds de l'arbre DOM

sont des objets Java. La Figure 4.13 montre l'utilisation du parseur DOM dans le processus de transformation d'une requête - réponse - XML en requête - réponse - Java et vise versa.

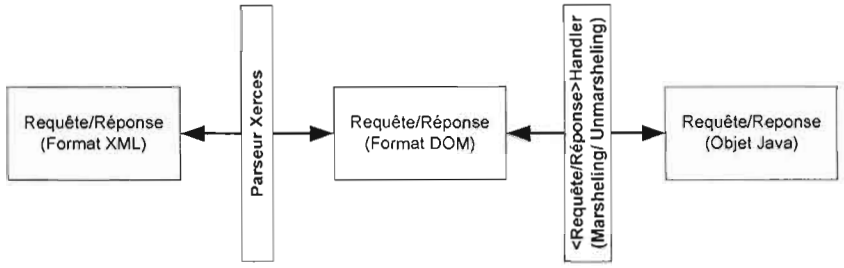

Figure 4.13. Utilisation d'un parseur DOM

4.3.4. Structure de la fonction exécutant la requête sur le registre jUDDI

Les requêtes reçues par le registre jUDDI sont exécutées chacune par une fonction qui la représente. Chaque fonction est représentée par une classe Java qui hérite de la classe **`AbstractFunction`** et qui implémente l'interface **`IFunction`**. Cette interface contient une seule méthode qui s'exécute en se basant sur le type de la requête reçue en paramètre et qui retourne une réponse qui dépend de cette requête. Par exemple, la classe **`FindServiceFunction`** présentée dans la Figure 4.14 décrit la fonction qui exécute la requête **`find_service()`**. Sa méthode **`execute()`** prend en paramètre un objet de type **`FindService`** et retourne comme réponse un objet de type **`ServiceList`**.

```
public class FindServiceFunction implements IFunction extends
             AbstractFunction {
...
  public RegistryObject execute(RegistryObject regObject)
                                        throws RegistryException {

         FindService request = (FindService) regObject;
          ...
         ServiceList list = new ServiceList();
         // Accomplie l'exécution et retourne la réponse
         return list;
      }
}
```

Figure 4.14. Class Java de la fonction qui exécute la requête find_service()

4.3.5. Les répartiteurs HandlerMaker et FunctionMaker

Comme nous l'avons déjà mentionné dans la section 1.3.1, les deux éléments **`HandlerMaker`** et **`FunctionMaker`** servent comme répartiteurs. L'élément **`HandlerMaker`** créé une correspondance entre les requêtes - les réponses – et les *handlers* qui lui sont associés. À partir du nom de la requête - sa réponse - sous format Java ou sous format DOM (XML), le *handler* associé est retourné (Figure 4.15). L'élément **`FunctionMaker`** associe une requête à la fonction qui l'exécute au niveau du registre (Figure 4.15). Toute requête envoyée au registre jUDDI passe par ces deux répartiteurs.

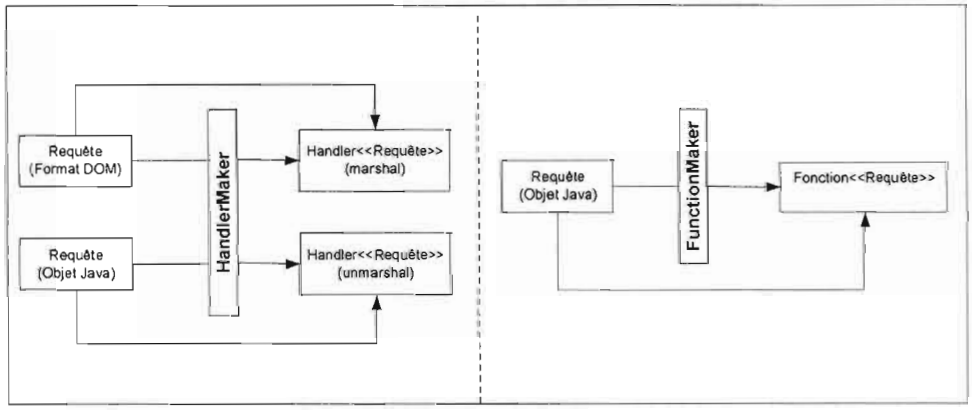

Figure 4.15. Les classes HandlerMaker et FunctionMaker

Dans la section suivante, nous présenterons l'architecture détaillée du registre JRegistre ainsi que les éléments jUDDI que nous allons réutiliser ou que nous allons étendre dans l'implémentation de notre registre.

4.4. JRegistre : architecture détaillée

Nous avons opté, dans l'implantation de JRegistre et de son proxy, pour la réutilisation du proxy jUDDI (Figure 4.16). Nous utilisons ce dernier comme noyau de notre registre. Ceci nous permet de réutiliser le code qui assure le processus d'envoi des requêtes et de réception des réponses UDDI standard au niveau du proxy JRegistre ainsi que le code du processus de transfert des requêtes UDDI standard du registre JRegistre au registre jUDDI (Figure 4.16). De plus, nous l'utiliserons pour assurer le transport des nouvelles requêtes/réponses entre proxy JRegistre et JRegistre (Figure 4.26). Notre travail consiste, en première phase, à ajouter

une couche au dessus du proxy jUDDI afin d'assurer le processus d'envoi et de réception de nouvelles requêtes/réponses au niveau du proxy JRegistre. En seconde phase, il y a lieu d'ajouter, au niveau de JRegistre, un module, au dessus du proxy jUDDI, qui permet la réception et l'envoi des requêtes/réponses. Ce module contient aussi la logique fonctionnelle des nouvelles requêtes (Figure 4.16). Ainsi, chaque requête reçue par JRegistre est traitée selon son type. Si la requête reçue est de type standard alors elle sera transférée au registre jUDDI via le proxy jUDDI. Sinon, un traitement local est effectué, et une séquence de requêtes UDDI standard seront envoyées au registre jUDDI via le proxy jUDDI.

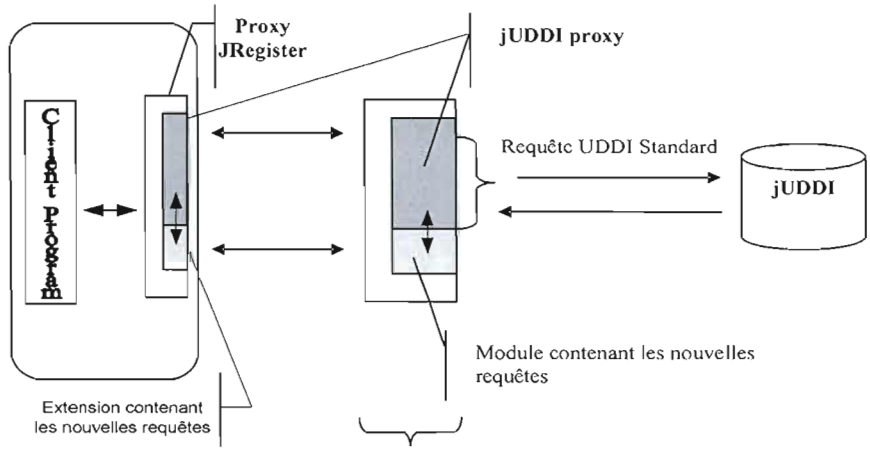

Figure 4.16. Ajout d'une couche au dessus du proxy jUDDI

Dans la section suivante, nous décrivons en détail les modifications apportées au processus d'envoi et de réception des requêtes/réponses entre le proxy et le registre.

4.4.1. Modifications apportées au processus d'envoi et de réception des requêtes/réponses

Afin de permettre l'échange de nouveaux types de requêtes/réponses entre le proxy JRegistre et le registre JRegistre, nous avons étendu quelques éléments du proxy jUDDI des deux cotés. Ci-après, les ajouts du côté proxy JRegistre :

- La classe **`JRegistreProxy`** est une extension de la classe **`JRegistryProxy`** contenue dans le proxy jUDDI. Cette nouvelle classe permet d'invoquer tout type de requête qu'elle soit de type standard ou de type complexe.

- La deuxième étape consiste à mettre en place un nouveau répartiteur pour faire une correspondance entre les nouvelles requêtes/réponses et les *handlers* qui lui sont associés. La classe **`JRHandlerMaker`** permet de trouver le *handler* associé à une requête ou à une réponse.

La Figure 4.17 montre les modifications apportées au processus d'envoi et de réception de requête/réponse du coté du proxy JRegistre.

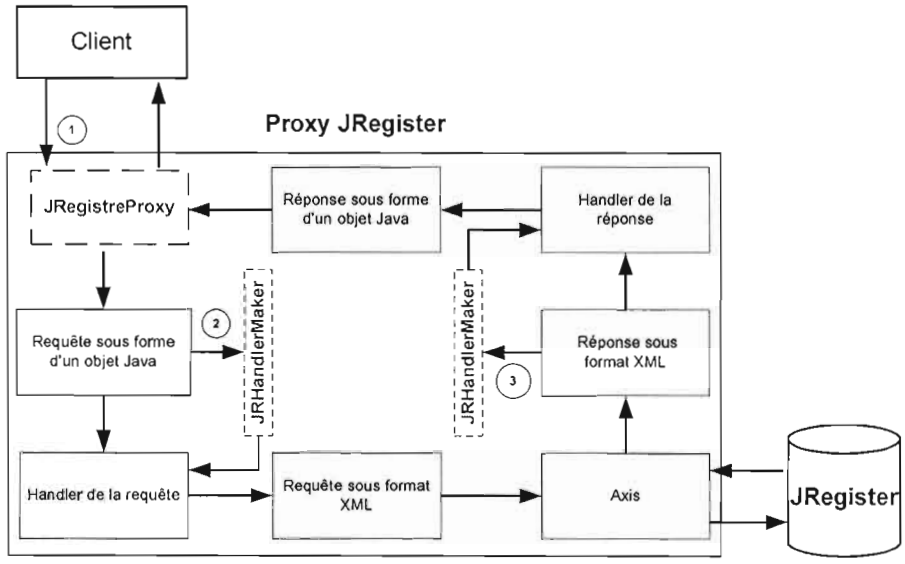

Figure 4.17. Processus d'envoi et de réception d'une requête/réponse (proxy)

Du côté registre, nous avons effectué quelques modifications au processus de réception de requêtes et d'envoi de réponses afin de permettre l'échange de nouveaux types de requêtes. Voici les modifications qui ont été apportées au processus d'échanges du côté registre :

a. De la même façon que du coté proxy JRegistre, nous avons mis en place un répartiteur qui permet de faire correspondre une requête reçue ou une

réponse à envoyer au *handler* qui lui est associé. La classe **`JRHandlerMaker`**, identique à celle du coté proxy, permet de réaliser cette tâche.

b. Nous avons ajouté un répartiteur qui permet d'associer une requête à la fonction qui l'exécute. La classe **`JRFunctionMaker`** associe toute requête connue par JRegistre à une fonction. Nous nous somme inspiré de la classe **`FunctionMaker`** existant au niveau de jUDDI pour développer notre classe.

La Figure 4.18 montre les modifications apportées au processus d'envoi et de réception de requête/réponse du côté du proxy JRegistre.

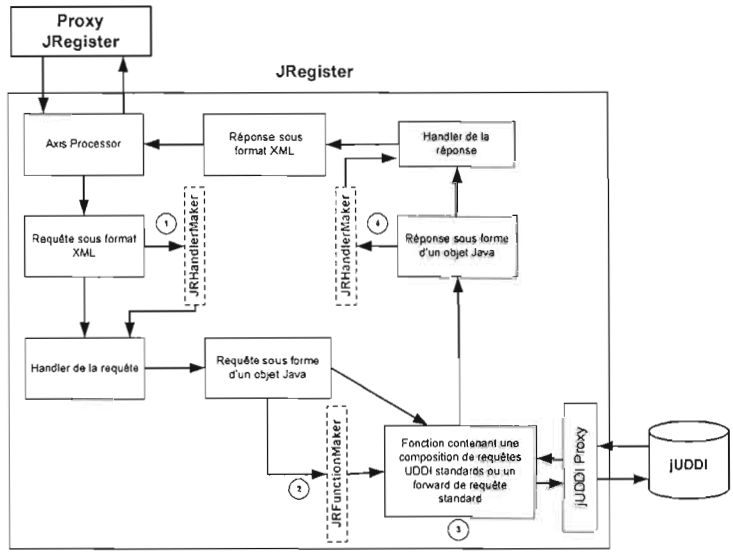

Figure 4.18. Processus de réception et d'envoi d'une requête/réponse (JRegistre)

Nous allons voir dans ce qui suit, les détails des modifications décrites ci-dessous.

4.4.2. Nouvelle interface pour les requêtes

Pour permettre l'invocation de nouvelles requêtes, nous avons étendu la façade d'invocation fournie dans le proxy jUDDI. La nouvelle façade permettra d'invoquer l'ensemble des requêtes UDDI standard ainsi que les nouvelles requêtes qui seront ajoutées au fil de l'utilisation du registre. La Figure 4.19 décrit cette nouvelle façade.

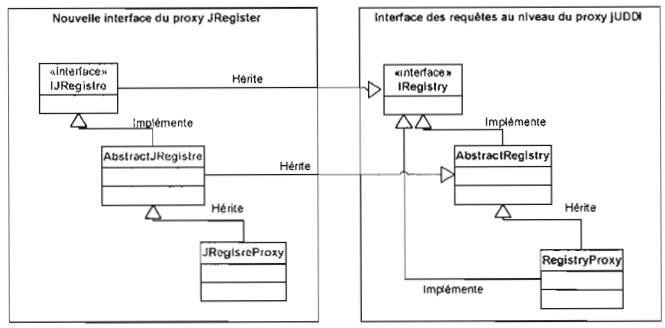

Figure 4.19. Façade d'invocation des requêtes

L'interface **`IJRegistre`** hérite de l'interface **`IRegistry`** (Figure 4.19) l'ensemble des méthodes qui permettent d'invoquer les requêtes UDDI standard. Cette interface contiendra plus tard les méthodes qui permettront d'invoquer les nouvelles requêtes ajoutées. La classe **`AbstractJregistre`** (Figure 4.19) implémente l'ensemble des méthodes définies dans l'interface **`IJRegistre`**. Ces méthodes permettent de construire l'objet représentant la requête, pour ensuite faire appelle à la méthode **`execute()`** définie dans la classe **`JRegistryProxy`** (Figure 4.19) qui permet de se connecter au registre, d'envoyer la requête à ce dernier et d'en recevoir la réponse.

Nous allons voir, plus loin dans la section 5.4.2, comment une nouvelle méthode représentant une requête est ajoutée à l'interface **`IJRegistre`** et à la classe **`AbstractJRegistre`**.

4.4.3. Ajout d'un nouveau répartiteur pour les handlers

Le nouveau répartiteur pour les *handlers* sert à associer l'ensemble des requêtes/réponses compréhensibles par JRegistre aux *handlers* correspondants. Nous avons essayé dans notre implémentation d'étendre la classe **`HandlerMaker`** définie dans le proxy jUDDI. Malheureusement, cela a été impossible. En effet, la classe **`HandlerMaker`** suit le patron de conception *Singleton* (référence). Une classe de ce type ne peut être héritée. Notre solution de remplacement était de créer la classe **`JRHandlerMaker`** avec sa propre table de correspondance <<requête, *handler*>>. Par la suite nous avons ajouté à cette classe, la méthode **`lookUp()`** afin de permettre de trouver le *handler* correspondant à une requête ou

une réponse connue sur JRegistre. Cette méthode délègue une partie de son travail à la méthode `lookUp()` de la classe `HandlerMaker`. (Figure 4.20)

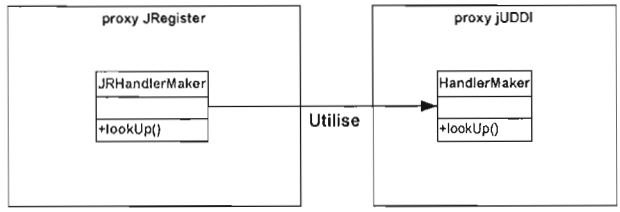

Figure 4.20. La classe JRHandlerMaker

Nous verrons en section 5.4.2, comment une nouvelle requête et une nouvelle réponse sont associées aux *handlers* au niveau de la classe `JRHandlerMaker`.

4.4.4. Ajout d'un nouveau répartiteur pour les fonctions

Le nouveau répartiteur permet d'associer tout type de requête à la fonction qui l'exécute. La particularité de ce répartiteur c'est qu'il permet de traiter les requêtes UDDI standard et les autres requêtes qui seront ajoutées au fil du temps. Si la requête est de type UDDI standard alors le répartiteur la dirige vers une fonction qui va la transférer telle quelle au registre jUDDI. Autrement, la requête est dirigée vers une fonction qui va la décomposer en un ensemble de requêtes UDDI standard pour les envoyer au registre jUDDI. La classe `JRFunctionMaker` (Figure 4.21) est ajoutée pour assurer cette tâche. Elle contient une table de correspondance « requête, fonction » et une méthode `lookUp()` qui retourne la fonction correspondant à une requête. Rappelons que nous nous sommes inspirés, dans l'implémentation de cette classe, de la classe `FunctionMaker` développée dans jUDDI.

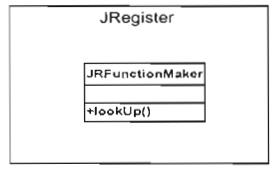

Figure 4.21. La classe JRFunctionMaker

La section 5.4.2 montrera comment une nouvelle requête est associée à la fonction qui l'exécute au niveau de la classe `JRFunctionMaker`.

4.5. Génération de requêtes complexes

À partir de l'analyse détaillée du processus d'échange entre le proxy jUDDI et le registre jUDDI vue précédemment, et en nous basant sur le code source de ces deux composants offert par la fondation Apache, nous avons pu extraire les éléments indispensables au bon fonctionnement de ce processus. Rappelons que le registre « JRegistre » doit suivre le même modèle d'échange que celui de jUDDI. Ainsi, il est nécessaire de mettre l'accent sur les éléments qui doivent être générés afin qu'une nouvelle requête soit ajoutée. Ci-après, un aperçu de toutes les étapes prises en compte dans la génération d'une requête/réponse :

- *Spécifier le format XML de la requête et de la réponse :* consiste à trouver une façon de décrire le format XML de la requête et de la réponse en se basant sur le même style UDDI. Une fois la requête et sa réponse spécifiées, on les envoie à un analyseur pour validation et chargement dans une structure DOM.

- *Générer une classe Java pour chaque requête et chaque réponse :* À ce niveau, il y a lieu de générer le format Java de la requête - ou de la réponse – à partir du format XML de la requête - ou de la réponse -. Une requête ou une réponse de type complexe peut donner naissance à plusieurs classes au niveau du registre. Cependant, une seule classe dite principale doit représenter la requête.

- *Générer une classe Java qui représente le handler pour chaque requête et pour chaque réponse :* le *handler* doit être capable de transformer la requête - ou la réponse - dans les deux sens; c'est-à-dire, du format Java au format XML (Marshaling) et du format XML au format Java (Unmarshaling). Une requête - ou une réponse - de type complexe peut avoir plus d'une classe qui représente son *handler.*

- *Générer la logique fonctionnelle derrière la requête :* la génération de la classe représentant la logique fonctionnelle sera assez facile dans la mesure où la fonction ne fera que des appels à des fonctions standard (find_business,

getBusinessDetail, etc.) et qu'elle n'impliquera pas des opérations de bas niveau (authentification, accès à la base de données, etc.).

Nous voulons que les opérations précédentes se fassent de façon automatique. Idéalement, toutes les classes Java devraient être générées, à la volée, durant l'exécution, et ce, à partir des descriptions des requêtes/réponses.

4.5.1. Spécification des requêtes/réponses basée sur les schémas XML

La technologie XML offre deux possibilités pour la modélisation de structure XML. Les DTD et les schémas XML, nommés aussi XSD (voir section 2.2.2). Nous avons opté, dans notre approche, pour la deuxième possibilité, et cela, pour plusieurs raisons. En effet, la spécification UDDI utilise XSD pour définir les requêtes/réponses standard. Il est donc préférable d'utiliser le même standard de modélisation pour rester conforme aux pratiques de l'industrie. De plus, les schémas XML nous fournissent la capacité de décrire le contenu des requêtes et des réponses de façon claire, flexible et compréhensible aussi bien par les humains que par les machines.

En nous basant sur les schémas XML des requêtes/réponses standard définis dans le document de spécification UDDI, nous avons pu déterminer un ensemble de règles pour la modélisation d'une requête complexe et de sa réponse. Ces règles sont sous forme de questions qui orientent le client publicateur de requêtes vers une modélisation optimale de sa requête. Voici la liste des questions possibles :

- En quoi consiste le traitement derrière la requête qu'on doit définir ? s'agit-il d'une composition de requêtes UDDI standard ou d'un traitement additionnel ? La logique fonctionnelle de la requête est écrite en se basant sur les réponses à ces questions.

- La structure de la requête est-elle simple ou complexe ? Il ne faut pas confondre entre une requête complexe et une requête ayant une structure complexe. Une requête complexe veut dire que le traitement derrière la

requête est complexe et qu'elle ne possède pas nécessairement une structure complexe. Il est possible d'avoir une requête ayant une structure simple, mais qui nécessite un traitement complexe. La complexité de la structure de la requête dépend de la nature de ses entrées. Ainsi, une requête ayant en entrée de simples chaînes de caractères est considérée de *structure simple* alors qu'une requête ayant en entrée des structures complexes est dite de *structure complexe*. Notons qu'une requête ayant des entrées de structures hybrides (simples et complexes) est considérée de structure complexe.

- La requête utilise-t-elle des éléments déjà définis dans UDDI ? Une requête peut intégrer dans sa structure un élément qui est déjà défini dans la spécification UDDI. UDDI offre un schéma XML qui définit l'ensemble des requêtes et des réponses standard. Deux alternatives sont possibles : i) copier la structure de l'élément qu'on veut intégrer et l'ajouter dans le schéma définissant la nouvelle requête ou ii) faire directement référence à la structure de l'élément qu'on veut intégrer. La deuxième alternative semble plus intéressante dans la mesure où elle permet d'appliquer l'aspect réutilisation assuré par notre approche. En effet, en faisant référence à l'élément au niveau du schéma XML de la spécification UDDI, nous pouvons, non seulement réutiliser la structure de cet élément au format XML, mais aussi réutiliser la structure Java qui le représente au niveau du registre jUDDI.

Les deux dernières questions doivent aussi être posées pour la modélisation de la réponse. Nous nous proposons, dès à présent, d'appliquer les questions décrites ci-dessus sur un exemple concret que nous avons défini. La requête `find_wsdl()` permet de trouver un ensemble d'URLs de documents WSDL en faisant une recherche par nom de service Web ou par nom de document WSDL. Cette requête est une composition d'un ensemble de requêtes UDDI standard (Figure 4.22). La requête `find_wsdl()` sera équivalente à la séquence suivante :

1. On applique une requête, de type **`find_service()`**, qui retourne une liste de services Web vérifiant certains critères.

2. Pour chaque service Web retourné, on applique une requête de type **`get_serviceDetail()`**, qui retourne les détails du service contenant l'ensemble des **`bindingTemplates`** qui lui sont liés.

3. Pour chaque **`bindingTemplate`**, on applique une requête de type **`get_bindingDetail()`**, qui retourne les détails de liaison et une référence vers un **`tModel`** qui, à sont tour, contient un lien vers un document WSDL.

4. On applique une requête de type **`get_tModelDetail()`**, qui retourne les détails du **`tModel`** contenant la référence vers le document WSDL du service.

5. On extrait le nom et l'URL du document WSDL du **`tModel`** et on l'ajoute à une liste qui contient le résultat.

Le scénario décrit dans la Figure 4.22 permet de rassembler toutes les informations nécessaires pour la construction de la logique fonctionnelle de la requête.

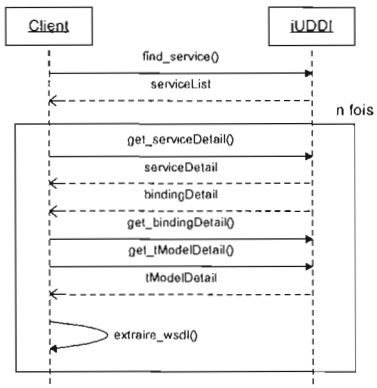

Figure 4.22. L'ensemble des requêtes UDDI standard équivalentes à find_wsdl()

La deuxième étape consiste à déterminer la structure de la requête. Autrement dit, les éléments d'entrée nécessaires et la structure de chacun d'entre eux. La requête **find_wsdl()** prend comme entrée un nom de service (**serviceName**) et un nom de WSDL (**wsdlName**) tous deux sous forme de chaînes de caractères. La Figure 4.23 montre la structure de la requête sous forme d'un schéma XML.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema targetNamespace="Jregistre.uqam.ca"
            xmlns="Jregistre.uqam.ca"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema">

    <xsd:complexType name="find_wsdl">
       <xsd:sequence>
          <xsd:element name="serviceName" type="xsd:string"/>
          <xsd:element name="wsdlName" type="xsd:string" />
       </xsd:sequence>
    </xsd:complexType>
</xsd:schema>
```

Figure 4.23. Schéma XML de la requête find_wsdl()

La spécification UDDI offre un **findQualifiers** qui permet de formater le résultat selon plusieurs critères et dans un ordre spécifié par l'utilisateur. Nous avons intégré cette option dans la requête (Figure 4.24). En faisant référence au **findQualifiers**, le schéma XML de la requête devient comme suit :

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema targetNamespace="Jregistre.uqam.ca"
            xmlns="Jregistre.uqam.ca"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:uddi="urn:uddi-org:api_v2">
    <xsd:complexType name="find_wsdl">
       <xsd:sequence>
          <xsd:element name="serviceName" type="xsd:string" />
          <xsd:element name="wsdlName" type="xsd:string" />
          <xsd:element name="qualifiers" type=uddi:findQualifiers"/>
       </xsd:sequence>
    </xsd:complexType>
</xsd:schema>
```

Figure 4.24. Schéma de la requête find_wsdl() faisant référence à un élément UDDI

L'étape suivante consiste à déterminer la structure de la réponse à la requête. La réponse est une liste d'éléments **wsdl**. Un **wsdl** est une structure complexe composée de

deux éléments : un mon de **wsdl** (**name**) et un lien vers le document **wsdl** (**url**). Ces deux éléments sont sous forme d'une chaîne de caractères.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema targetNamespace="Jregistre.uqam.ca"
            xmlns="Jregistre.uqam.ca"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema" >
<xsd:complexType name="wsdl_list">
        <xsd:sequence>
            <xsd:element name="wsdl" minOccurs="1">
                <xsd:complexType name="WSDL">
                   <xsd:sequence>
                        <xsd:element name="name" type="xsd:string"/>
                        <xsd:element name="url" type="xsd:string"/>
                    </xsd:sequence>
                </xsd:complexType>
             </xsd:element>
        </xsd:sequence>
    </xsd:complexType>
</xsd:schema>
```

Figure 4.25. Schéma XML de la réponse à la requête find_wsdl()

Une fois que la requête et sa réponse sont spécifiées sous format XSD, on les envoie comme éléments d'entrées au générateur de requêtes. Nous présentons, dans les sections qui suivent, les étapes de génération d'une requête et de sa réponse.

4.5.2. Analyse des schémas XML représentant la requête/réponse

Pour générer les classes Java représentant une requête, une réponse et les *handlers* associés, il y a lieu d'analyser les schémas XML (XSD) qui décrivent la requête et sa réponse. Ensuite, il faut les charger dans une arborescence qui permettra de les manipuler, et cela dans le but d'accéder aux informations qui, à leur tour, permettront de générer les classes représentant la requête, la réponse, et les *handlers* associés. A cet effet, nous avons utilisés le parseur XSD offert par EMF™ (*Eclipse Modeling Framework*) qui permet de manipuler des documents XSD (EMF-XSD, 2002). Cependant, le *parsing* de EMF n'est pas assez complet pour collecter les informations nécessaires à la génération. En effet, le parseur permet d'analyser les types de bases et les types complexes définis localement, mais ne permet pas d'analyser les types externes référencés dans le schéma de la requête ou de la réponse. En particulier, le parseur ne permet pas d'analyser les éléments UDDI référencés dans le schéma d'une requête ou d'une réponse. Nous introduisons, dans notre approche, une représentation

intermédiaire pour collecter les informations nécessaires pour la phase de génération. La Figure 4.26 montre un aperçu du modèle proposé pour la représentation intermédiaire d'une requête ou d'une réponse. Nous introduisons, dans ce modèle, une hiérarchie de concepts qui permettent de définir clairement la structure d'une requête ou d'une réponse. La classe racine **`JRType`** décrit les types de données qui sont connus par le registre JRegistre. La classe **`JRBasicType`** représente tous les types de bases (int, float, String,...). La classe **`JRClass`**, quant à elle, représente tous les types complexes représentés par une classe Java. L'expression « types complexes » désigne toutes les nouvelles structures complexes définies dans JRegistre ainsi que les types prédéfinis dans jUDDI. Pour différencier ces deux groupes, nous introduisons la classe **`JRPredefinedClass`** pour présenter les types prédéfinis de jUDDI et la classe **`JRUserClass`** pour présenter les types complexes de JRegistre. Par ailleurs, nous considérons qu'une nouvelle requête ou une nouvelle réponse est une classe Java qui contient un ensemble d'attributs (**`JRAttribute`**). Chaque attribut a un type. Le type peut être de base ou complexe.

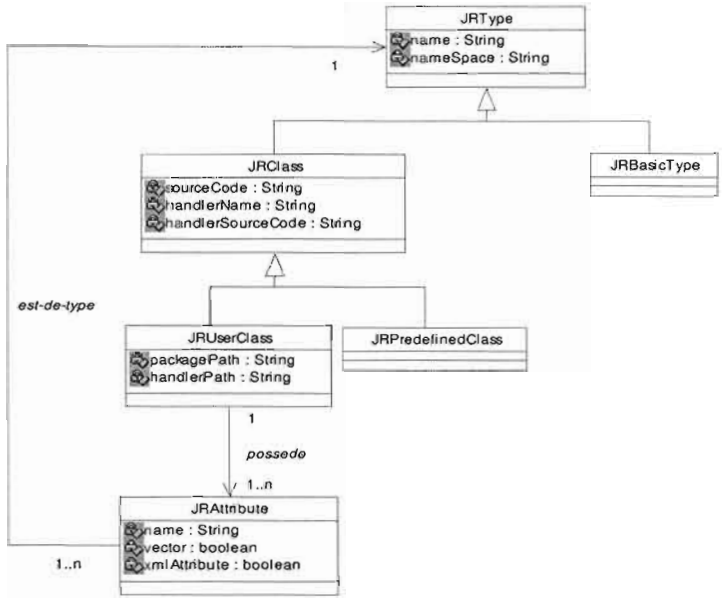

Figure 4.26. Modèle intermédiaire de représentation d'une requête/réponse

L'exemple suivant montre une requête **`find_wsdl()`** (Figure 4.27) composée de trois éléments : deux éléments de types string et un troisième élément qui fait référence à un type UDDI.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema targetNamespace="Jregistre.uqam.ca"
            xmlns="Jregistre.uqam.ca"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema"
            xmlns:uddi="urn:uddi-org:api_v2">

    <xsd:complexType name="find_wsdl">
       <xsd:sequence>
          <xsd:element name="serviceName" type="xsd:string" />
          <xsd:element name="wsdlName" type="xsd:string" />
          <xsd:element name="qualifiers" type=uddi:findQualifiers"/>
       </xsd:sequence>
    </xsd:complexType>
</xsd:schema>
```

Figure 4.27. Schéma XML de la requête find_wsdl()

La représentation de cette requête, selon le modèle intermédiaire proposé, ressemblera globalement à celle décrite dans la Figure 4.28. Il est important de noter que dans ce modèle intermédiaire, les types XML de bases sont transformés en des types Java de base. Par exemple, **`xsd:string`** est transformé en **`String`**. De même, le nom de la requête est transformé en un nom de classe et cela en se basant sur les normes Java et jUDDI. A titre d'exemple, **`find_wsdl`** est transformé en **`FindWsdl`**. Ce modèle permet aussi de transformer les types UDDI (XSD) en des types jUDDI (Java). Par exemple, **`uddi:findQalifiers`** est transformé en **`FindQalifiers`**. (Figure 4.28)

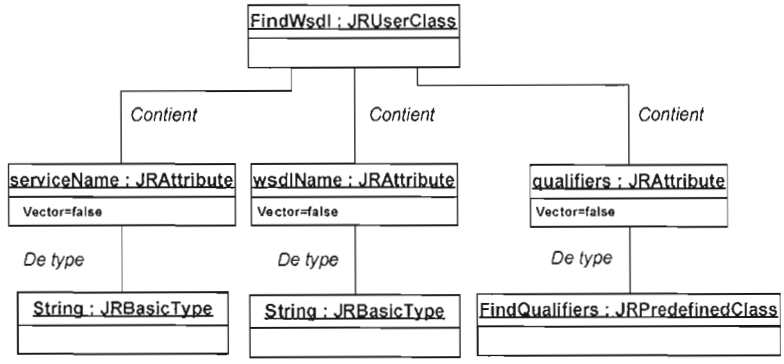

Figure 4.28. Représentation intermédiaire de la requête find_wsdl()

Une réponse est traitée de la même façon qu'une requête. La réponse **wsdl_list** (Figure 4.29) à la requête **find_wsdl(),** décrite précédemment, possède une structure plus complexe que celle de cette dernière. La réponse contient un élément **wsdl** (Figure 4.29) qui possède lui-même une structure complexe.

```
<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema targetNamespace="Jregistre.uqam.ca"
            xmlns="Jregistre.uqam.ca"
            xmlns:xsd="http://www.w3.org/2001/XMLSchema" >
<xsd:complexType name="wsdl_list">
        <xsd:sequence>
            <xsd:element name="wsdl" minOccurs="1">
                <xsd:complexType name="WSDL">
                   <xsd:sequence>
                        <xsd:element name="name" type="xsd:string"/>
                        <xsd:element name="url" type="xsd:string"/>
                    </xsd:sequence>
                </xsd:complexType>
             </xsd:element>
        </xsd:sequence>
    </xsd:complexType>
</xsd:schema>
```

Figure 4.29. Schéma XML de la réponse wsdl_list

La représentation de cette réponse, selon le modèle intermédiaire proposé, est alors différente de celle de la requête. La Figure 4.30 montre globalement la structure intermédiaire de la réponse **wsdl_list**.

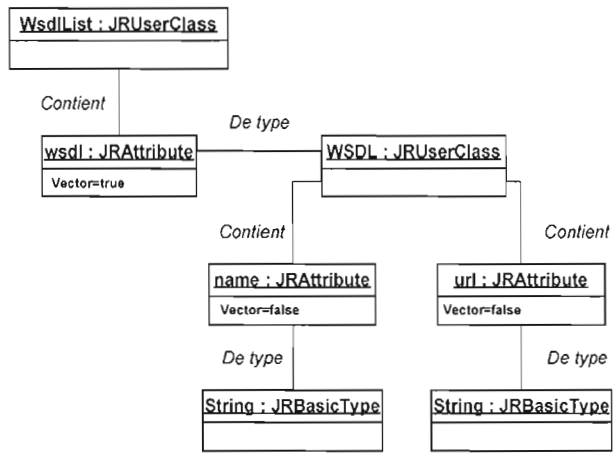

Figure 4.30. Représentation intermédiaire de la réponse wsdl_list

4.5.3. Génération des classes représentant la requête et sa réponse

En nous basant sur le modèle intermédiaire de la requête et de celui de la réponse, défini dans la section précédente, nous générons un ensemble de classes Java qui représente la requête et sa réponse au niveau du registre. L'ensemble des classes générées varie selon la structure intermédiaire de la requête et de sa réponse. Que ça soit pour la requête ou pour la réponse, nous avons énuméré un ensemble de règles qui définissent l'étape de génération des classes les représentants :

- l'élément racine de la structure intermédiaire, qui est de type `JRUserClass`, correspond à une nouvelle classe qui porte le même nom que la racine. Cette classe implémente l'interface `RegistryObject`. Notons, que `RegistryObject` est une interface de jUDDI qui représente le parent de toutes les requêtes et les réponses jUDDI.

- Chaque élément de type `JRUserClass` correspond à une nouvelle classe qui porte le même nom.

- Chaque élément de type `JRPredefinedClass` correspond à un type jUDDI déjà défini.

- Chaque élément de type `JRBasicType` correspondra à un type de base Java.

- Les éléments de type `JRAttribute`, faisant partie d'un élément de type `JRUserClass`, correspondront à des attributs de la classe portant le même nom que cet élément.

Les informations contenues dans la représentation intermédiaire sont prises comme telles sans qu'aucune modification n'y soit apportée. Rappelons que les transformations (transformation de noms XML vers Java, ...), selon le standard Java et jUDDI, ont été appliquées au moment de la création de cette représentation intermédiaire.

Pour illustrer globalement le processus de génération de classes représentant la requête et de sa réponse à partir de la représentation intermédiaire, nous allons reprendre l'exemple de la requête **`find_wsdl()`** donnée en (Figure 4.28). La Figure 4.31 décrit la mise en correspondance entre la structure intermédiaire de cette requête et la classe qui doit être générée. L'élément **`FindWsdl`** racine de la structure intermédiaire, de type **`JRUserClass`**, correspond à une nouvelle classe qui porte le même nom. Cette classe représente la requête **`find_wsdl()`** au niveau du registre JRegistre. Les éléments **`serviceName`**, **`wsdlName`** et **`qualifiers`** correspondent à des attributs de la classe **`FindWsdl`**. Les deux premiers ont un type de base (**`JRBasicType`**) et l'élément **`qualifiers`** est de type **`JRPredefinedClass`**, donc un type déjà défini dans jUDDI. On fait référence dans la classe **`FindWsdl`** au type de l'attribut **`qualifiers`**.

Selon les normes de développement appliquées par les réalisateurs de jUDDI, les classes représentant une requête - réponse - contiennent un ensemble d'opérations qui permettent de manipuler les attributs de ces classes. Pour rester dans le sillage du même style de programmation et pour rester conforme à jUDDI, nous avons intégré un mécanisme de génération automatique de méthodes pour la manipulation des attributs des nouvelles classes générées.

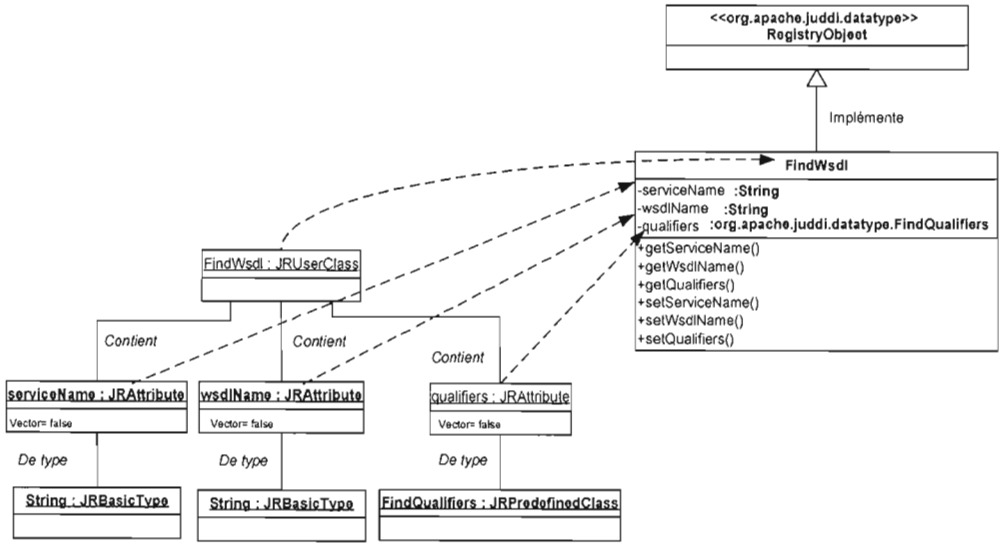

Figure 4.31. Génération de la classe Java représentant la requête find_wsdl()

De même que pour la requête, la génération de la réponse suit le même procédé. La Figure 4.32 décrit la mise en correspondance entre la représentation intermédiaire de la réponse **`WsdlList`** et les classes qui doivent être générées.

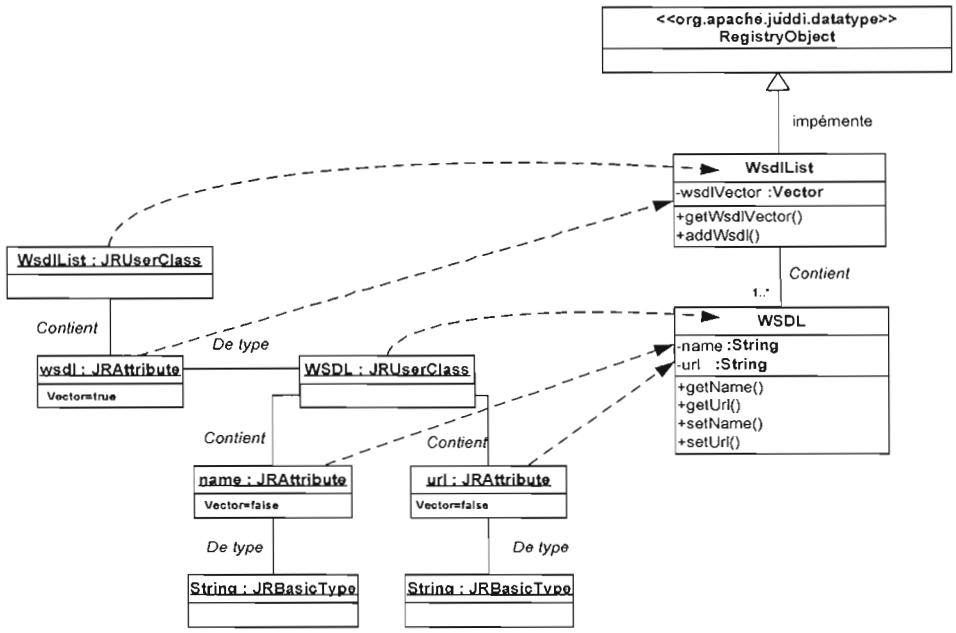

Figure 4.32. Génération des classes Java représentant la réponse Wsdl_list

Comme suite à la génération, des classes représentant la requête et sa réponse, vient l'étape de génération des *handlers*. On rappelle que les *handlers* sont responsable de la transformation d'une requête - réponse - XML en un objet Java qui la représente et vice-versa. Nous décrivons dans la section suivante le processus détaillé de génération des *handlers*.

4.5.4. Génération des classes représentant les handlers de la requête et de sa réponse

En nous basant sur le modèle intermédiaire de la requête et de celui de la réponse, nous générons un ensemble de classes Java qui représente les *handlers* pour la requête et pour sa réponse au niveau de JRegistre. Pour chaque classe générée représentant une requête - réponse -, on produit une classe capable de transformer un objet Java, de la classe générée, en un élément DOM et vice-versa. Que ça soit pour la requête ou pour la réponse, nous

énumérons, dans ce qui suit, un ensemble de règles qui définissent l'étape de génération des classes représentants les *handlers* :

- Pour chaque élément de type **`JRUserClass`**, on génère une classe. Par exemple pour la réponse **`wsdl_list`** on génère deux classes **`WsdlListHandler`** et **`WSDLHandler`** qui représentent le *handler* de cette réponse (Figure 4.32).

- Chaque classe contient deux opérations :

  - L'opération « marshal » qui prend un objet Java et qui le transforme en un élément DOM. Par exemple, l'opération marshal de la classe **`WsdlListHandler`**, prend un objet Java de type **`WsdlList`** et le transforme en un élément DOM représentant la requête **`wsdl_list`**.

  - L'opération « unmarshal » qui prend un élément DOM et retourne un objet Java. Par exemple, l'opération unmarshal de la classe **`WsdlListHandler`**, prend un élément DOM représentant la requête **`wsdl_list`** et retourne un objet Java de type **`WsdlList`**.

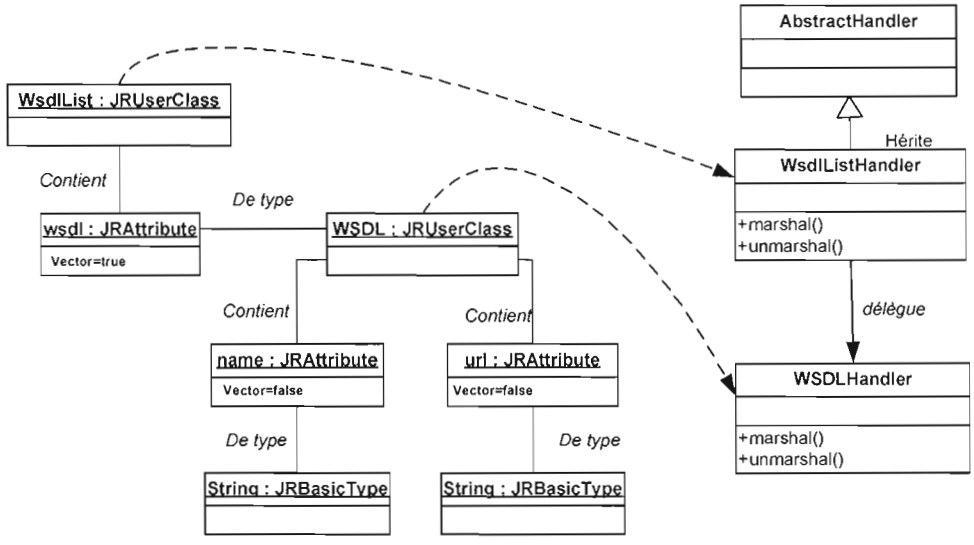

Figure 4.33. Génération des classes représentant le handler de la réponse Wsdl_list

4.5.5. Génération de la fonction exécutant la requête

Dans notre approche, chaque requête doit posséder une fonction qui s'occupe de son exécution au niveau de JRegistre. Cette fonction est une classe Java ayant le même nom que la requête. Cette classe doit implémenter obligatoirement l'interface `IFunction` parent de toutes les fonctions de JRegistre. Notons, au passage, que l'interface `IFunction` est identique à celle définie dans jUDDI. Elle contient une seule méthode qui prend en entrée un objet requête et qui retourne un objet réponse. Par exemple, la classe `FindWsdlFunction`, représente la fonction qui exécute la requête `find_wsdl()`. La méthode `execute()` de cette classe prend en entrée un objet de type `FindWsdl` représentant la requête et retourne un objet de type `WsdlList` qui représente la réponse. Dans le cadre de notre travail, nous avons proposé trois solutions de génération de la fonction exécutant la requête. Nous en présentons l'aperçu dans ce qui suit :

- *Solution 1 :* Génération d'une classe Java représentant la fonction à partir d'une description d'un processus d'affaires écrite dans un langage tel que BPEL4WS (BPEL4WS, 2002).

- *Solution 2 :* Génération d'une classe Java qui délègue l'exécution à un interpréteur BPEL capable d'exécuter un processus BPEL4WS.

- *Solution 3 :* Développement de la classe manuellement. Dans ce cas, le développeur écrit code la fonction, en dur, puis l'intègre au niveau du registre.

Pour ne pas dévier du but de notre sujet et vu que les deux premières solutions constituent des axes de recherche en soit, nous avons décidé de nous intéresser à la troisième solution. Les deux autres peuvent pouvant être abordées dans des travaux futurs.

4.6. Conclusion

Dans ce chapitre, nous avons présenté l'architecture que nous proposons pour le registre JRegistre, et expliqué les étapes nécessaires pour générer une nouvelle requête sur ce dernier. La génération des classes Java représentant la requête, sa réponse et leurs *handlers* ne suffit à la rendre opérationnelle. Pour ce faire, il y a lieu de mettre à jour plusieurs

composantes de JRegistre et ce, à chaque fois qu'une nouvelle requête est ajoutée. Nous allons voir dans le chapitre suivant comment ces mises à jour sont réalisées pendant l'exécution

CHAPITRE V

Mise à jour dynamique

5.1. Mise à jour dynamique

Rappelons que notre approche se distingue par le besoin de pouvoir faire les changements de façon dynamique. Elle s'appuie sur l'aspect dynamicité du registre. En effet, l'idée de modifier le code source manuellement et de redémarrer le registre chaque fois qu'un nouveau type de requête est ajouté ne constitue pas une solution acceptable. Notre solution consiste en une extension aux registres UDDI qui supporte l'ajout dynamique de nouvelles requêtes complexes. Ainsi, l'ajout d'un nouveau type de requête à notre registre par une entreprise quelconque doit rendre cette requête visible immédiatement aux autres clients autorisés. En d'autres termes, on vise à ce que l'intégration d'une nouvelle requête au registre se fasse dynamiquement, sans intervention humaine et obligation de réinitialiser le système.

Techniquement, une compagnie X qui veut publier un nouveau type de requête, doit envoyer au générateur de requêtes sous JRegistre : i) la description de la requête, ii) la description de sa réponse, et iii) la description de la logique fonctionnelle régissant la requête. Une fois ces données analysées, le générateur produit un ensemble de classes qui représentent la requête, sa réponse, les *handlers* et la logique fonctionnelle d'exécution de la requête (voir chapitre IV pour plus de détails). Ces classes sont ensuite compilées et introduites dans l'archive qui contient l'ensemble des requêtes de JRegistre. À ce niveau, la requête est présente sur le registre. Cependant, elle n'est pas encore fonctionnelle. En effet, des mises à jour doivent être effectuées sur certains composants du registre pour activer le processus de renvoi et de réception de la requête ajoutée. De plus, les clients qui veulent utiliser ce type de requête doivent mettre à jour le proxy JRegistre qu'ils utilisent. Nous allons voir dans le reste

de ce chapitre une description détaillée des exigences requises ainsi que les solutions retenues pour résoudre le problème de la dynamicité.

5.2. Exigences/problèmes de conception

5.2.1. L'ajout de support pour une nouvelle requête n'implique pas que des ajouts de classes

La génération des classes qui représentent une nouvelle requête/réponse ne suffit pas pour rendre la requête fonctionnelle au niveau de JRegistre. Plusieurs composantes de notre registre doivent être mises à jour chaque fois qu'une nouvelle requête est ajoutée. Du code source doit être injecté dans chacune des classes ou interface qui représentent ces composantes. Voici un aperçu de ce qui doit être mis à jour :

- La liste des requêtes offertes au niveau de l'interface **`IJRegistre`** et de la classe **`AbstractJRegistre`**. On veut ajouter automatiquement une signature de méthode représentant la requête au niveau de l'interface **`IJRegistre`** et une méthode au niveau de la classe **`AbstractJRegistre`**. Par exemple, supposons qu'on veut ajouter la requête **`find_wsdl()`** au registre, alors on doit ajouter la signature **`findWsdl(`*`paramètres`*`)`** au niveau de l'interface **`IJRegistre`** et la méthode **`findWsdl(paramètres){...}`** au niveau de la classe **`AbstractJRegistre`**. Ces ajouts vont permettre d'adjoindre cette nouvelle requête à la liste des requêtes offertes par JRegistre.

- La classe **`JRHandlerMaker`** pour créer une correspondance entre la nouvelle requête et son *handler*, et entre la nouvelle réponse et son *handler*. Par exemple, pour la requête **`find_wsdl()`**, on doit créer une correspondance entre la requête **`find_wsdl()`** et le *handler* **`FindWsdlHandler`** et une correspondance entre la réponse **`WsdlList`** et le *handler* **`WsdlListHandler`**.

- La classe **`JRFunctionMaker`** pour créer une correspondance entre la requête et sa logique fonctionnelle. Par exemple, pour la requête **`find_wsdl()`**, on

doit créer une correspondance entre la requête `find_wsdl()` et la fonction `FindWsdlFunction`.

Nous présenterons dans la section 5.3.1 la solution que nous proposerons pour résoudre ce problème.

5.2.2. Mise à jour des clients et serveur de JRegistre

L'ajout d'une nouvelle requête au registre nécessite des mises à jour du côté registre ainsi que du côté proxy (client). Nous constatons que ces mises à jour sont presque identiques des deux côtés. Par exemple, supposons qu'on veuille ajouter la requête `find_wsdl()` au registre. Les classes représentant cette nouvelle requête, sa réponse et leur *handlers* doivent être ajoutées au registre et au proxy. De plus, des mises à jour sur l'interface `IJRegistre` et les classes `AbstractJRegistre` et `JRHandlerMaker` doivent s'effectuer du côté clients (proxy) ainsi que du côté serveur JRegistre, chaque fois qu'une nouvelle requête est ajoutée. Devons-nous effectuer le même travail deux fois ? Est-ce qu'on doit faire des mises à jour du côté registre et ensuite les refaire sur le proxy ? Nous présenterons dans la section 5.3.2 la solution que nous proposerons pour résoudre ce problème.

5.2.3. Opérationnalisation dynamique d'une requête

Java supporte le chargement dynamique de classes. Une application peut charger une classe qui n'était pas chargée au lancement. En effet, la machine virtuelle de Java offre un mécanisme de chargement de classes sur demande. Seules les classes utilisées sont chargées. Quand une classe est chargée (loaded) pour la première fois, son code binaire est copié (cached) en mémoire cache. Cela permet de minimiser le temps de chargement de ces classes si elles sont utilisées antérieurement. De ce fait, si une classe est modifiée au cours de l'exécution, les modifications ne seront pas opérationnelles immédiatement puisque le chargeur de classes (classloader) va chercher d'abord, s'il y a une version en mémoire cache. Les serveurs Web comme Tomcat ont résolu ce problème en intégrant une méthode de chargement par session. Chaque client possède une session qui utilise son propre chargeur (classloader). Par conséquent, un client qui démarre une session après une modification

effectuée sur un ensemble de classes aura la nouvelle version des classes puisque le chargement se fait automatiquement chaque fois qu'une nouvelle session client est démarrée.

Dans le cas de notre projet, nous voulons que les nouvelles requêtes ajoutées par les clients soient chargées immédiatement, et cela sans être obligé de redémarrer le registre. Puisque JRegistre est déployé sur le serveur Tomcat, le chargement des nouvelles classes qui représentent une nouvelle requête/réponse et le rechargement des classes modifiées après l'ajout d'une nouvelle requête sont alors assurés par le chargeur (classloader) du serveur Tomcat lui-même. Une nouvelle requête est automatiquement opérationnelle au démarrage d'une session ultérieure, soit pour le client lui-même soit pour les autres utilisateurs.

5.3. Principe de la solution

5.3.1. Utilisation de la programmation orientée aspects pour la modification automatique du code source

Pour résoudre le problème de mise à jour des classes décrites dans la section 5.2.1, nous avons eu recours à la programmation orientée aspects. Nous allons voir en détail la description de cette technique dans la section 5.4.1. Ce qui nous intéresse dans la programmation orientée aspect, c'est qu'elle permet de modifier le comportement d'une classe en lui ajoutant du comportement à des points d'exécution particuliers. Nous avons utilisé l'outil AspectJ, une implémentation publique de la fondation Eclipse pour modifier les classes `IJRegistre`, `AbtractJRegistre`, `JRHandlerMaker` et `JRFunctionMaker` chaque fois qu'une nouvelle requête est ajoutée. Dans la section 5.4.3, nous présenterons les détails des modifications apportées à ces classes.

5.3.2. Une archive en commun

Pour résoudre le problème décrit dans la section 5.2.2, nous avons structuré les archives du côté registre ainsi que du côté proxy de façon à mettre la partie commune dans une archive à part. Cette archive, appelée `jar-commons-part`, contiendra l'ensemble des classes représentant les requêtes présentes sur le registre, leurs réponses et leurs *handlers*. Elle contiendra aussi l'interface `IJRegistre` et les classes `AbtractJRegistre` et

`JRHandlerMaker`. Du fait que l'archive `jar-commons-part` est commune aux parties client (proxy) et serveur JRegistre, il suffit de deployer cette archive des deux côtés.

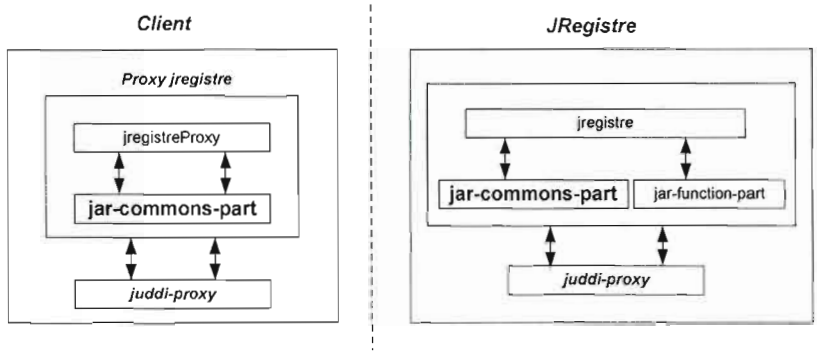

Figure 5.1. Structure des archives au niveau du registre et du proxy

5.3.3. Fonctionnement du générateur de requêtes

Dans la section 4.5, nous avons expliqué la fonctionnalité du générateur de requête (le «quoi »). Dans cette section, nous expliquerons le « comment », au regard des contraintes de conception décrites dans la section 5.2. La Figure 5.2 décrit l'architecture du générateur de requêtes. Rappelons qu'un client qui veut publier un nouveau type de requêtes, doit envoyer la description de la requête et la description de sa réponse sous format XSD via une interface de publication Web au générateur de requêtes. Une fois les descriptions reçues au niveau du générateur, elles sont traitées par le servlet `JRRequestPublisherServlet`. Notons que le servlet `JRRequestPublisherServlet` est l'élément racine responsable de l'exécution du processus de génération et d'intégration d'un nouveau type de requêtes. Nous présenterons ci-bas, les étapes du traitement requis pour l'ajout d'une nouvelle requête.

- *Étape 1* : Les descriptions de la requête et de sa réponse sont stockées au niveau de JRegistre. L'élément `JRListner` enregistre la description de la requête et de sa réponse dans deux fichiers (*request.xsd* et *response.xsd*). Cette étape est nécessaire pour la suite (Figure 5.2).

- *Étape 2* : À partir des deux fichiers créés à l'étape précédente, l'élément `JRGenProcess` génère le code source de l'ensemble des classes Java qui vont

représenter la requête, sa réponse et leurs *handlers* au niveau de JRegistre (Figure 5.2).

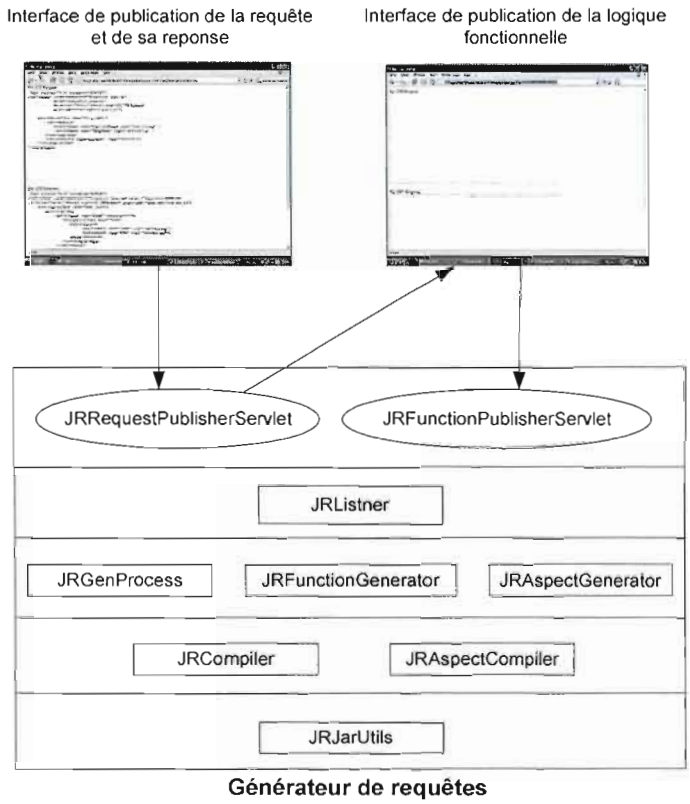

Figure 5.2. Architecture du générateur de requêtes

- *Étape 3* : Générer le squelette de la fonction exécutant la requête. Cette étape aide le client publicateur de la requête à développer cette fonction. L'élément `JRFunctionGenerator` permet de générer cette fonction à partir du couple requête/réponse .

- *Étape 4 :* Compiler l'ensemble des classes générées aux étapes 2 et 3 (composant `JRCompiler`) et les ajouter aux archives correspondantes (composant `JRJarUtils`). Notez que les nouvelles classes sont déployées dans deux archives différentes, i) l'archive `jar-commons-part` contenant l'ensemble des requêtes, leurs réponses, et leurs *handlers*, et qui devra être

déployée tant du coté serveur que du coté client, et ii) l'archive **jar-function-part**, qui contient les classes représentant les fonctions qui exécutent les requêtes, et qui n'est déployée que du coté du serveur.

- *Étape 5* : modifier les classes d'aiguillage (dispatching) du JRegistre (**IJRegistre**, **AbstractJRegistre**, **JRHandlerMaker**, et **JRFunctionMaker**) pour tenir compte du nouveau type de requête, en utilisant la programmation par aspects, (composant **JRAspectCompiler**) et remplacer les versions compilées dans les archives correspondantes (**IJRegistre**, **AbstractJRegistre**, **JRHandlerMaker**, dans **jar-commons-part**, et **JRFunctionMaker** dans l'archive **jar-function-part**).

- *Étape 6 :* Mettre à jour les anciennes archives du registre. L'élément **JRJarUtils** copie les deux nouvelles archives dans le répertoire */lib* qui contient les anciennes archives. Cette action permet de recharger les classes qui ont été mises à jour à l'étape 5 sans redémarrer le registre.

- *Étape 7 :* Mettre à jour le proxy. L'élément **JRJarUtils** copie l'archive **jar-commons-part** dans le répertoire */proxy*. Tous les clients qui veulent avoir un registre à jour doivent télécharger cette nouvelle archive de ce répertoire.

Jusque-là, la requête est totalement intégrée au registre. Cependant, la logique fonctionnelle qui prend en charge l'exécution de cette requête est encore incomplète. Le générateur de requête renvoie au client publicateur de la requête, un projet eclipse qui contient le squelette de la logique fonctionnelle et les deux archives **jar-function-part** et **jar-commons-part** nécessaires pour le développement de cette fonction. Le publicateur développe cette fonction manuellement et l'envoie via une interface de publication. Cette interface communique directement avec le servlet **JRFunctionPublisherServlet** qui va réaliser les étapes suivantes :

- *Étape 1 :* Compiler la nouvelle logique fonctionnelle. L'élément `JRCompiler` compile la nouvelle classe qui représente la logique fonctionnelle. En cas d'erreur de compilation, cet élément retourne le message d'erreur généré par le compilateur Java standard (Figure 5.2).

- *Étape 2 :* Remplacer le squelette de la fonction par la nouvelle fonction. L'élément `JRJarUtils` remplace l'ancienne classe contenant le squelette par la classe contenant la nouvelle fonction dans l'archive `jar-funtion-part`. L'action est effectuée sur une copie du jar pour éviter d'endommager le jar en cas de problème (Figure 5.2).

- *Étape 3 :* Mettre à jour l'archive. L'élément `JRJarUtils` copie l'archive résultant de l'étape précédente et le met dans le répertoire */lib*. Cette action permet de recharger la nouvelle logique fonctionnelle sans redémarrer le registre (Figure 5.2).

Une fois ces étapes accomplies, la nouvelle requête est fonctionnelle. Nous présenterons dans l'annexe A, un extrait détaillé et documenté du code source du servlet `JRRequestPublisherServlet` et du servlet `JRFunctionPublisherServlet`.

5.3.4. Mises à jour requises pour les clients

Le client publicateur de la requête et les autres clients qui veulent utiliser une nouvelle requête ajoutée au registre doivent mettre à jour leur copie de l'archive `jar-commons-part`. Nous avons étudié plusieurs alternatives en nous basant sur les préférences des clients, la faisabilité et la simplicité d'implantation. Nous en sommes arrivés à deux solutions :

- L'installation d'une application du côté client et qui permet de télécharger la version la plus récente de l'archive `jar-commons-part` ferait le travail. Cette application va chercher cette archive dans le répertoire */proxy* présent du côté registre.

- Aller télécharger manuellemement cet archive sur le site Web du registre.

Nous avons retenu la première solution.

5.4. Utilisation de AspectJ

5.4.1. Programmation orientée aspect et AspectJ

La programmation orientée aspects a été introduite en 1997. C'est un paradigme de programmation qui permet de réduire le couplage entre les différents aspects techniques d'une application et la fonctionnalité d'affaires (Kiczales et al., 1997). Kiczales et al. considèrent une application comme un ensemble de définitions et de propriétés. Chacune de ces propriétés peut être définie dans un module indépendant appelé « aspect ». Selon Kiczales et al., une application peut être décomposée en trois parties :

- *Un aspect de base* qui définit l'ensemble des exigences fonctionnelles de l'application. Par exemple, dans un système de gestion de commande, « Faire une commande » est considéré comme une exigence fonctionnelle.

- *Un ensemble d'aspects non fonctionnels* qui définissent les exigences non fonctionnelles. Chaque aspect est indépendant des autres. Par exemple, la persistance et la sécurité sont considérées comme des exigences non fonctionnelles.

- *Un compositeur d'aspects* permettant de composer les aspects décrits ci-dessus afin de construire l'application. La composition de l'aspect de base avec les aspects non fonctionnels se traduit par l'établissement d'une jonction entre ces aspects. Cette jonction s'établit au niveau d'un ensemble de points du flot d'exécution de l'aspect de base, appelés *points de jonction* (Kiczales et al., 1997). Une application est alors le résultat d'un tissage de plusieurs modules indépendants qui représentent les définitions de différents aspects fonctionnels et non fonctionnels.

Depuis l'apparition de la programmation orientée aspect, plusieurs travaux ont essayé de mettre cette technique en pratique. AspectJ est une extension Java qui implémente la programmation orientée aspect (Kiczales et al., 2001). Il représente la première solution qui intègre toutes les spécifications de cette technique.

En AspectJ, les aspects sont des entités au même titre que des classes : Ils peuvent être re-utilisés, étendus, et manipulés de la même façon que les classes. AspectJ représente les points de jonction par un couple de (*coupe*, *action*). La *coupe* (*pointcut*) correspond à la définition syntaxique d'un ensemble de points de jonction et l'*action* (*advice*) spécifie le code à exécuter à ces points de jonction. Les coupes peuvent se présenter sous plusieurs formes : dont l'appel d'une méthode ou d'un constructeur, l'initialisation d'un objet, le lancement d'une exception, etc. Il existe trois types d'actions, les « *before advices* », les « *around advices* » et les « *after advices* ». Ces derniers permettent de spécifier si l'action doit s'exécuter avant ou/et après les points de jonction. Pour comprendre le principe du couple (*coupe*, *action*) introduit dans AspectJ, nous allons prendre l'exemple décrit dans la Figure 5.3 auquel on appliquera l'aspect de la Figure 5.4.

```
1 public class HelloWorld {
2
3    public static void main(String[] args) {
4     HelloWorld h = new HelloWorld();
5     h.hello();
6    }
7    public void hello() {
8           System.out.println(" Hello World");
9    }
```

Figure 5.3. La classe HelloWorld

```
1 public aspect HelloWorldAspect {
2
3   pointcut taratata(): call(public void hello()) ;
4
5   before() : taratata (){
6             System.out.println("before");
7   after() : taratata (){
8              System.out.println("after");
9
10            }
11    }
```

Figure 5.4. Exemple d'un aspect écrit en AspectJ

La ligne 3 de la Figure 5.4 définit une coupe (*pointcut*) nommée « **taratata()** » qui spécifie un point de jonction au niveau de l'appel de la méthode ayant la signature

suivante «**public void hello()**» (Figure 5.3). Les lignes 5 et 7 de la Figure 5.4 définissent les actions qui doivent être déclenchées.

L'application d'un aspect à une classe se fait à l'aide du compilateur *ajc (aspect java compiler)* d'aspectJ . Le compilateur génère à partir des deux entrées (classe et aspect) soit une source .Java ou un fichier compilé .class. Il est possible d'appliquer un aspect sur un fichier source .Java, un .class ou même sur un jar qui contient une classe donnée. L'application de l'aspect **HelloWorldAspect** sur la classe **HelloWorld.java** donne alors une source Java plus ou moins semblable à celle de la Figure 5.5.

```
 1 public class HelloWorld {
 2
 3    public static void main(String[] args) {
 4     HelloWorld h = new HelloWorld();
 5     h.hello();
 6
 7    }
 8    public void hello() {

 9           System.out.println(" before");
10           System.out.println(" Hello World");
11             System.out.println(" after");

12    }
```

Figure 5.5. Résultat de l'application de l'aspect HelloWorldAspect

5.4.2. Intégration du code en utilisant AspectJ

a) Ajout d'une nouvelle requête/réponse à la façade du registre (IJRegistre et AbstractJRegistre) par application d'aspect

L'application d'aspect au niveau de la façade consiste à ajouter une signature de méthode représentant la requête au niveau de l'interface **IJRegistre** et une méthode au niveau de la classe **AbstractJRegistre**. Cette méthode contient le code nécessaire pour invoquer la requête et pour recevoir sa réponse. Nous présentons dans la Figure 5.6, un exemple d'aspect qui permet d'ajouter la requête **find_wsdl()** à la façade du registre.

```
package ca.uqam.jregistre.aspect;

import ca.uqam.jregistre.AbstractJRegistre;
import ca.uqam.jregistre.IJRegistre;
import org.apache.juddi.error.RegistryException;
import ca.uqam.jregistre.datatype.response.wsdllist.WsdlList;
import ca.uqam.jregistre.datatype.request.findwsdl.FindWsdl ;

public aspect JRFindWsdlAspect{

// Permet d'ajouter la signature de la méthode à IJRegistre

public abstract WsdlList ca.uqam.jregistre.IJRegistre.findWsdl
                    (int maxRows,String serviceName,String WsdlName)
                                          throws RegistryException;

// Permet d'ajouter la méthode représentant la requête find_wsdl() à
la classe AbstractJRegistre

public WsdlList ca.uqam.jregistre.AbstractJRegistre.findWsdl
                    (int maxRows,String serviceName,String WsdlName)
                                          throws RegistryException
{
 FindWsdl request = new FindWsdl();
 request.setMaxRows(maxRows);
 request.setServiceName(serviceName);
 request.setWsdlName(WsdlName);
 return (WsdlList)execute(request);
}
}
```

Figure 5.6. Ajout de la requête find_wsdl() à la façade du registre

b) Ajout des handlers de la requête/réponse au JRHandlerMaker par application d'aspects

L'application d'aspects sur la classe **`JRHandlerMaker`** aura pour conséquence l'ajout de deux entrées dans cette classe pour faire un lien entre la requête et son *handler* et la réponse et son *handler*. Autrement dit, l'application de l'aspect intégrera deux méthodes dans la classe **`JRHandlerMaker`** qui permettront de construire le lien entre la requête/réponse et les *handlers* qui lui sont associés et d'ajouter cette association dans la table qui contient toutes les associations. La Figure 5.7 décrit un exemple d'aspect qui permet d'intégrer une telle association.

```
package ca.uqam.jregistre.aspect;

import ca.uqam.jregistre.handler.JRHandlerMaker;
…
public aspect JRFindWsdlAspect{

// Methode qui permet d'ajouter le handler de la requête find_wsdl()
au JRHandlerMaker

void ca.uqam.jregistre.handler.JRHandlerMaker.$addFindWsdlHandler()
{JRHandlerMaker.handler=new FindWsdlHandler(this);

// Ajoute l'association de la requête sous format java à son handler
dans la table qui contient toutes les associations.

JRHandlerMaker.handlers.put(FindWsdl.class.getName().toLowerCase(),J
RHandlerMaker.handler);

// Ajoute l'association de la requête sous format XML à son handler
dans la table qui contient toutes les associations.

JRHandlerMaker.handlers.put(FindWsdlHandler.TAG_NAME.toLowerCase(),J
RHandlerMaker.handler);
}
// Permet d'ajouter le handler de la réponse wsdl_list au
JRHandlerMaker

void ca.uqam.jregistre.handler.JRHandlerMaker.$addWsdlListHandler()
{
JRHandlerMaker.handler=new WsdlListHandler(this);

// Ajoute l'association de la requête sous format java à son handler
dans la table qui contient toutes les associations.

JRHandlerMaker.handlers.put(WsdlList.class.getName().toLowerCase(),J
RHandlerMaker.handler);
// Ajoute l'association de la requête sous format XML à son handler
dans la table qui contient toutes les associations.

JRHandlerMaker.handlers.put(WsdlListHandler.TAG_NAME.toLowerCase(),J
RHandlerMaker.handler);}
}
```

Figure 5.7. Ajout des handlers de la nouvelle requête/réponse au JRHandlerMaker

c) Mise à jour du JRFunctionMaker

L'application d'aspects sur la classe **JRFunctionMaker** aura pour conséquence l'ajout d'une entrée dans cette classe qui créera un lien entre la requête et la fonction qui

l'exécute au niveau du registre. La Figure 5.8 présente un exemple d'aspect permettant de réaliser cet ajout.

```
package ca.uqam.jregistre.aspect;

import ca.uqam.jregistre.function.JRFunctionMaker;
...
public aspect JRFindWsdlFunctionAspect{

// Methode qui permet d'ajouter la fonction exécutant la requête
find_wsdl() au JRFunctionMaker

void ca.uqam.jregistre.function.JRFunctionMaker.$addFindWsdlFunction
()
{JRFunctionMaker.function=new FindWsdlFunction();

// Ajoute l'association de la requête à sa fonction qui l'exécute
dans la table qui contient toutes les associations.

JRFunctionMaker.functions.put(FindWsdl.class.getName(),JRFunctionMak
er.function);}
}
```

Figure 5.8. Ajout de la fonction exécutant la requête au JRFunctionMaker

5.5. Conclusion

L'intégration d'une nouvelle requête au registre de façon dynamique représente une exigence fonctionnelle inévitable. L'étape de génération de code représentant une requête, décrite dans le chapitre précèdent, ne suffisait pas à rendre celle-ci fonctionnelle au niveau de JRegistre. Plusieurs composantes doivent être mises à jour de façon automatique et dynamique pour tenir compte de l'ajout d'une nouvelle requête. Pour ce faire, nous avons utilisé AspectJ, une technique basée sur la programmation orientée aspect pour mettre à jour certaines composantes du registre afin qu'ils reconnaissent une nouvelle requête ajoutée. Plus précisément, nous avons utilisé aspectJ pour modifier les classes **`IJRegistre`**, **`AbsractJRegistre`**, **`JRHandlerMaker`**, **`JRFunctionMaker`**, qui jouent un rôle clé dans le processus d'envoi et de réception des requêtes. Nous avons utilisé aussi le mécanisme de chargement dynamique de classes offert par le serveur Tomcat pour charger les nouvelles requêtes et recharger les composantes qui ont été mises à jour. La combinaison de ces deux solutions permet à une nouvelle requête de devenir opérationnelle immédiatement après son ajout au registre.

CHAPITRE VI

Conclusion

Les Services Web ont apporté une solution aux problèmes d'échanges de données et d'intégration d'applications entre entreprises. Encore faut-il que les entreprises puissent identifier ou sélectionner des partenaires d'affaires pour un besoin particulier, dans le contexte d'un marché électronique ouvert. La norme UDDI spécifie la structure et la fonctionnalité de registres ou *annuaires* de services, auprès desquels des entreprises peuvent enregistrer les services qu'elles offrent, et venir chercher des services dont elles ont besoin. Comme toute norme industrielle, la norme UDDI est un compromis entre idéalisme et pragmatisme. Idéalement, on aimerait pouvoir capturer le maximum d'informations sur les services offerts en terme de, i) sémantique des opérations, ii) contraintes d'utilisation, iii) qualité de service, iv) fiabilité des partenaires, et bien d'autres. De façon pragmatique, la mise en œuvre de la norme doit pouvoir se faire de façon aisée, et elle se doit donc de s'aligner le plus possible avec le « plus petit dénominateur commun » des descriptions de services que l'on peut retrouver en pratique.

Plusieurs chercheurs ont identifié un certain nombre d'utilisations de registres pour lesquelles la norme UDDI s'avère inadéquate. Ils ont alors proposé d'augmenter, et 1) la richesse des descriptions de services, et 2) les fonctionnalités de consultation de registres UDDI. Les approches proposées partagent le but commun de rendre le processus de découverte, de composition et d'invocation de service Web plus flexible. Certaines de ces approches ont proposé d'ajouter une couche sémantique à UDDI (Paolucci et al., 2003) et (Colgrave et al., 2004). D'autres approches ont proposé d'intégrer des modèles de découverte de services Web basés sur la qualité de service (Qos). L'inconvénient majeur de toutes ces approches réside dans le fait qu'elles nécessitent l'implantation d'un nouveau registre UDDI.

D'autres encore ont étendu le standard UDDI dans le but de faciliter la découverte et la composition dynamique de services Web (Zhang et al., 2003 B) et (Srivatava et Koehler., 2003). Ces approches utilisent un intermédiaire qui fait un traitement local pour ensuite déléguer une partie de son travail à un ou plusieurs registres UDDI standard. Ce type d'approche ne nécessite pas la réimplantation des registres existants, mais crée un nouveau problème : un client doit parler à différents APIs spécifiques à chacune des extensions qu'il veut utiliser (QoS, Composition dynamique, etc.), ce qui rend l'utilisation de ces extensions lourde.

Dans ce travail, notre objectif n'était pas d'étendre les registres UDDI pour répondre à un besoin spécifique, mais plutôt de proposer un cadre générique permettant d'ajouter de telles extensions, de façon uniforme, cohérente avec la norme, et à l'exécution, càd sans arrêt du registre. Grosso modo, notre solution consiste à fournir un registre intermédiaire qui supporte l'ajout de nouveaux types de requêtes, et qui agit comme un intermédiaire *unique* entre les clients et un ensemble de registres UDDI standards. De plus, cet ajout peut-être fait à l'exécution : il sera pris en compte immédiatement après, et deviendra accessible aux utilisateurs.

Concrètement, nous avons développé un registre intermédiaire, que nous avons appelé JRegistre, qui agit comme « courtier » entre les clients et les registres standards. Pour ajouter une nouvelle fonctionnalité d'interrogation, l'utilisateur doit soumettre à JRegistre, i) la structure de la requête, 2) la structure de la réponse à la requête, et 3) la logique d'exécution de la requête. Cette logique d'exécution qui sera exécutée sur JRegistre, se chargera de , i) traduire la nouvelle requête en des requêtes UUDI standards, ii) faire exécuter ces requêtes sur les registres UDDI standard, et iii) composer leurs résultats pour produire la réponse à la requête initiale. Notre implantation de JRegistre est basée sur l'architecture du registre jUDDI, une implantation Java de registres UDDI développée par la fondation Apache (jUDDI, 2003). Pour offrir l'ajout dynamique de requêtes, nous avons dû utiliser un certain nombre de techniques de programmation de pointe, dont la programmation par aspects (Kiczales et al., 1997).

Notre approche présente deux limitations majeures. La première est reliée à la description de la logique de calcul des nouvelles requêtes. Pour le moment, le client qui souhaite ajouter une nouvelle requête doit coder sa logique de calcul manuellement sous la forme d'une classe Java. Cette approche trahit un peu notre souci d'abstraction, et contraint le client à soumettre sa requête en deux étapes (voir section 5.3.3). Une meilleure approche consisterait à saisir la logique de calcul des requêtes de façon abstraite, et à générer le code Java correspondant en même temps que les requêtes. Un tel langage pourrait être le langage BPEL4WS (BPEL4WS,2002), étant donné que les registres UDDI standards auxquels on délèguera les requêtes standards sont eux-mêmes des services Web. Dans ce cas, nous n'aurions même pas besoin de générer du code Java équivalent : il suffirait de doter JRegistre d'un interpréteur BPEL4WS.

Une deuxième limitation de notre approche est liée aux types de données stockées sur des registres UDDI standard. Notre registre joue le rôle d'un courtier (*broker*) qui ne fait que traiter les données stockées ailleurs, i.e. dans des registres UDDI standards. Donc, JRegistre, par son mécanisme d'extension, ne permet pas de prendre en compte des données de qualité de service (QoS), par exemple. Une solution serait d'accepter que notre JRegistre puisse emmagasiner des données complémentaires à ce qui se trouve dans les registres UDDI standard. Ceci permettra d'élargir l'éventail d'extensions que notre cadre supporte, tout en rendant sa mise en œuvre moyennement plus complexe.

ANNEXE A
EXEMPLES DE CODE DE JREGISTRE

A) Code source jUDDI modifiés pour l'implantation de JRegistre

Comme nous l'avons déjà mentionné dans la section 4.4.1, l'échange de nouveau type de requêtes/réponses entre le proxy JRegistre et le registre JRegistre, nécessite l'extension de quelques éléments de jUDDI . Voici le code source des éléments étendus :

Code source de la classe **JRegistreProxy**

```
/* Created on Nov 15, 2004
 * this class represents all the UDDI request specified on UDDI 2.0
 * and all new request added on JRegistre. Note that we can add
 * dynamicly the JRegistre request
 * @author Radhouane Ben-tamrout
 * @version 1.01
 */
package ca.uqam.jregistre.proxy;

import java.net.URL;
import java.util.Properties;
import ca.uqam.jregistre.AbstractJRegistre;
import org.apache.juddi.datatype.RegistryObject;
import org.apache.juddi.error.RegistryException;
import ca.uqam.jregistre.handler.JRHandlerMaker;
import org.apache.juddi.handler.IHandler;
import org.apache.juddi.handler.HandlerMaker;
import org.apache.juddi.util.xml.XMLUtils;
import org.apache.juddi.proxy.Transport;
import org.apache.juddi.proxy.AxisTransport;
import org.w3c.dom.Document;
import org.w3c.dom.Element;

public class JRegistreProxy extends AbstractJRegistre
{
      private Properties proxyProperties = new Properties();
```

```
      private  JRHandlerMaker maker = new JRHandlerMaker();
      private  HandlerMaker juddiMaker = HandlerMaker.getInstance();
      // JRegistre Proxy Property Names
      private static final String ENDPOINT_PROPERTY_NAME =
"jregistre.proxy.url";
  //JRegistre version Property Names
      private static final String JREGISTRE_VERSION_PROPERTY_NAME =
"jregistre.version";

  //JRegistre nameSpace Property Names
      private static final String JREGISTRE_NAMESPACE_PROPERTY_NAME =
"jregistre.nameSpace";

  //used UDDI version Property Names
  public static final String UDDI_VERSION = "2.0";

  //UDDI nameSpace Property Names
  public static final String UDDI_NAMESPACE = "urn:uddi-org:api_v2";

  public static String JREGISTRE_VERSION = "1.0";

  // JRegistre nameSpace Property Names
  public static String JREGISTRE_NAMESPACE = "urn:jregistre-uqam-
ca:api_v1";

  //JRegistre default endPoint
  public static String
  DEFAULT_ENDPOINT="http://localhost:8080/jregistre/receiver";

  private URL endpoint=null;
  private String jregistreVersion=null;
  private String jregistreNameSpace = null;

  public JRegistreProxy(Properties props)
  {
    super();
    this.proxyProperties=props;
  }

  public RegistryObject execute(RegistryObject request)
    throws RegistryException
  {
      RegistryObject response=null;
      try
      {
    URL endPointURL = new URL(DEFAULT_ENDPOINT);
    if(proxyProperties.getProperty(ENDPOINT_PROPERTY_NAME)==null)
    {
      endPointURL = new
URL(proxyProperties.getProperty(ENDPOINT_PROPERTY_NAME));
    }
```

```
if(proxyProperties.getProperty(JREGISTRE_VERSION_PROPERTY_NAME)!=nul
l)
        {JREGISTRE_VERSION=
proxyProperties.getProperty(JREGISTRE_VERSION_PROPERTY_NAME);}

      if(proxyProperties.getProperty(JREGISTRE_NAMESPACE_PROPERTY_NAM
E)!=null)

{JREGISTRE_VERSION=proxyProperties.getProperty(JREGISTRE_NAMESPACE_P
ROPERTY_NAME);}

    // create a new 'temp' XML element. This
    // element is used as a container in which
    // to marshal the jregistre request into.

    Document document = XMLUtils.createDocument();
    Element temp = document.createElement("temp");

    // lookup the appropriate request handler
    // and marshal the RegistryObject into the
    // appropriate xml format

    String requestName = request.getClass().getName();
    IHandler requestHandler = maker.lookup(requestName);
    requestHandler.marshal(request,temp);
    Element req = (Element)temp.getFirstChild();

    // if the request is a simple UDDI request (UDDI 2.0) then add
    // the specific UDDI tag:
    // "xmlns" to specify that the request is a UDDI request
    // "generic" to specify the version of UDDI used

if(maker.getJRegistreHandlers().get(request.getClass().getName().toL
owerCase())==null)
    {
      req.setAttribute("generic",UDDI_VERSION);
          req.setAttribute("xmlns",UDDI_NAMESPACE);
      }
    else
    {
      req.setAttribute("generic",JREGISTRE_VERSION);
          req.setAttribute("xmlns",JREGISTRE_NAMESPACE);
          }

    // A SOAP request is made and a SOAP response is returned.

    Transport transport = new AxisTransport();
    Element res = transport.send(req,endPointURL,false,null,0);

    // First, let's make sure that a response
    // (any response) is found in the SOAP Body.
```

```
    String responseName = res.getLocalName();
    if (responseName == null)
    {
      throw new RegistryException("Unsupported response " +
        "from registry. A value was not present.");
    }
    // Let's now try to determine which jregistre response
    // we received and unmarshal it appropriately or
    // throw a RegistryException if it's unknown.

    IHandler handler = maker.lookup(responseName.toLowerCase());
    if (handler == null)
    {
      throw new RegistryException("Unsupported response " +
        "from registry. Response type '" + responseName +
        "' is unknown.");
    }
    // Well, we have now determined that something was
    // returned and it is "a something" that we know
    // about so let's unmarshal it into a RegistryObject

    response = handler.unmarshal(res);
    // Next, let's make sure we didn't recieve a SOAP
    // Fault. If it is a SOAP Fault then throw it
    // immediately.

    if (response instanceof RegistryException)
      throw ((RegistryException)response);

    // That's it. Return the response to the caller.
      }
      catch(Exception e){e.printStackTrace();}
    return response;
  }
}
```

Code source de l'interface **JRegistre**

```
/* Created on nov 12, 2005
 * This interface represents all the JRegistre request.
 * it will be modified by aspectJ to add a new request.
 * @author Radhouane Ben-tamrout
 * @version 1.01
 */
package ca.uqam.jregistre;
import org.apache.juddi.IRegistry;

public interface IJRegistre extends IRegistry {
}
```

Code source de la classe **AbstractJRegistre**

```
/**
 * Created on nov 12, 2005
 * @author Radhouane Ben-tamrout
 * @version 1.01
 * This class represents all the JRegistre request.
 * it will be modified by aspectJ to add a new request.
 */
package ca.uqam.jregistre;
import org.apache.juddi.AbstractRegistry;
import ca.uqam.jregistre.IJRegistre;
import org.apache.juddi.datatype.RegistryObject;
import org.apache.juddi.error.RegistryException;

public abstract class AbstractJRegistre extends AbstractRegistry
implements IJRegistre {

  public AbstractJRegistre() {
    super();
  }
  public abstract RegistryObject execute(RegistryObject object)
throws RegistryException;
}
```

Code source de la classe **JRHandlerMaker**

```
/**
 * Created on Dec 23, 2004
 * this class contains all the JRegistre and JUDDI handlers
 * it will be modified by aspectJ to add a new handlers
 * @author Radhouane Ben-tamrout
 * @version 1.01
 */

package ca.uqam.jregistre.handler;

import java.util.HashMap;
import org.apache.juddi.handler.AbstractHandler;
import org.apache.juddi.handler.HandlerMaker;
import ca.uqam.jregistre.reflection.JRInspection;

public class JRHandlerMaker {

      // this hashMap contains all the JRegistre handlers
      public static HashMap handlers;

      // JUDDI maker contains all the JUDDI handlers
```

```
    private static HandlerMaker maker = HandlerMaker.getInstance();

    // all handlers extends the <<AbstractHandler>>
    public static AbstractHandler handler;

     /**
     * This constructor of JRHandlerMaker
     */
    public JRHandlerMaker(){

          handlers= new HashMap();
        handler =null;

        // create a inspection instance

        JRInspection inspection = new JRInspection();

        // by reflexion invoke all special methods
        // these methods are added by AspectJ
        //  each method adds a handler on the handlers hashtable

        inspection.invokeAllSpecialMethods(this);
    }

    /** getters */

    public HashMap getJRegistreHandlers(){return handlers;}

     /**
     * this method return the handler by XML element name or
     * by  java object(JUDDI or JRegistre Object) name
     * @param elementName
     * @return AbstractHandler
     */
    public final AbstractHandler lookup(String elementName)
          {
           // element name keys are stored in lower
           // case only so we don't need to be concerned
           // about inconsistencies in SOAP providers

            String key = elementName.toLowerCase();
          handler = (AbstractHandler)handlers.get(key);

          // if the Object recieved is not a JRegistre object.
          // Then the object is a JUDDI object
          if (handler == null)
            {handler = maker.lookup(elementName);}
    return handler;
          }
}
```

Code source de la classe **JRFunctionMaker**

```
/**
 * Created on Dec 28, 2004
 * this class contains all the JRegistre and JUDDI functions
 * it will be modified by aspectJ to add a new functions
 * @author Radhouane Ben-tamrout
 * @version 1.01
 */

package ca.uqam.jregistre.function;

import java.util.HashMap;
import ca.uqam.jregistre.reflection.JRInspection;

import org.apache.juddi.datatype.request.AddPublisherAssertions;
import org.apache.juddi.datatype.request.DeleteBinding;
import org.apache.juddi.datatype.request.DeleteBusiness;
import org.apache.juddi.datatype.request.DeletePublisher;
import org.apache.juddi.datatype.request.DeletePublisherAssertions;
import org.apache.juddi.datatype.request.DeleteService;
import org.apache.juddi.datatype.request.DeleteTModel;
import org.apache.juddi.datatype.request.DiscardAuthToken;
import org.apache.juddi.datatype.request.FindBinding;
import org.apache.juddi.datatype.request.FindBusiness;
import org.apache.juddi.datatype.request.FindPublisher;
import org.apache.juddi.datatype.request.FindRelatedBusinesses;
import org.apache.juddi.datatype.request.FindService;
import org.apache.juddi.datatype.request.FindTModel;
import org.apache.juddi.datatype.request.GetAssertionStatusReport;
import org.apache.juddi.datatype.request.GetAuthToken;
import org.apache.juddi.datatype.request.GetBindingDetail;
import org.apache.juddi.datatype.request.GetBusinessDetail;
import org.apache.juddi.datatype.request.GetBusinessDetailExt;
import org.apache.juddi.datatype.request.GetPublisherAssertions;
import org.apache.juddi.datatype.request.GetPublisherDetail;
import org.apache.juddi.datatype.request.GetRegisteredInfo;
import org.apache.juddi.datatype.request.GetRegistryInfo;
import org.apache.juddi.datatype.request.GetServiceDetail;
import org.apache.juddi.datatype.request.GetTModelDetail;
import org.apache.juddi.datatype.request.SaveBinding;
import org.apache.juddi.datatype.request.SaveBusiness;
import org.apache.juddi.datatype.request.SavePublisher;
import org.apache.juddi.datatype.request.SaveService;
import org.apache.juddi.datatype.request.SaveTModel;
import org.apache.juddi.datatype.request.SetPublisherAssertions;
import org.apache.juddi.datatype.request.ValidateValues;
import java.lang.reflect.*;

public class JRFunctionMaker
{
  public static HashMap functions = new HashMap();
```

```
 public static IFunction function = null;

/**
 * Constructor  of JRFunctionMaker
 *
 */


 public JRFunctionMaker()
 {
   // put all the forword JUDDI standard functions on a hashtable

   function = new AddPublisherAssertionsFunction();
   functions.put(AddPublisherAssertions.class.getName(),function);

   function = new DeleteBindingFunction();
   functions.put(DeleteBinding.class.getName(),function);

   function = new DeleteBusinessFunction();
   functions.put(DeleteBusiness.class.getName(),function);

   function = new DeletePublisherAssertionsFunction();
 functions.put(DeletePublisherAssertions.class.getName(),function);

   function = new DeletePublisherFunction();
   functions.put(DeletePublisher.class.getName(),function);
   function = new DeleteServiceFunction();
   functions.put(DeleteService.class.getName(),function);

   function = new DeleteTModelFunction();
   functions.put(DeleteTModel.class.getName(),function);

   function = new DiscardAuthTokenFunction();
   functions.put(DiscardAuthToken.class.getName(),function);

   function = new FindBindingFunction();
   functions.put(FindBinding.class.getName(),function);

   function = new FindBusinessFunction();
   functions.put(FindBusiness.class.getName(),function);

   function = new FindPublisherFunction();
   functions.put(FindPublisher.class.getName(),function);

   function = new FindRelatedBusinessesFunction();
   functions.put(FindRelatedBusinesses.class.getName(),function);

   function = new FindServiceFunction();
   functions.put(FindService.class.getName(),function);

   function = new FindTModelFunction();
   functions.put(FindTModel.class.getName(),function);
```

```
 function = new GetAssertionStatusReportFunction();
functions.put(GetAssertionStatusReport.class.getName(),function);

 function = new GetAuthTokenFunction();
 functions.put(GetAuthToken.class.getName(),function);

 function = new GetBindingDetailFunction();
 functions.put(GetBindingDetail.class.getName(),function);

 function = new GetBusinessDetailFunction();
 functions.put(GetBusinessDetail.class.getName(),function);

 function = new GetBusinessDetailExtFunction();
 functions.put(GetBusinessDetailExt.class.getName(),function);

 function = new GetPublisherAssertionsFunction();
 functions.put(GetPublisherAssertions.class.getName(),function);

 function = new GetPublisherDetailFunction();
 functions.put(GetPublisherDetail.class.getName(),function);

 function = new GetRegisteredInfoFunction();
 functions.put(GetRegisteredInfo.class.getName(),function);

 function = new GetRegisteryInfoFunction();
 functions.put(GetRegistryInfo.class.getName(),function);

 function = new GetServiceDetailFunction();
 functions.put(GetServiceDetail.class.getName(),function);

 function = new GetTModelDetailFunction();
 functions.put(GetTModelDetail.class.getName(),function);

 function = new SaveBindingFunction();
 functions.put(SaveBinding.class.getName(),function);

 function = new SaveBusinessFunction();
 functions.put(SaveBusiness.class.getName(),function);

 function = new SavePublisherFunction();
 functions.put(SavePublisher.class.getName(),function);

 function = new SaveServiceFunction();
 functions.put(SaveService.class.getName(),function);

 function = new SaveTModelFunction();
 functions.put(SaveTModel.class.getName(),function);

 function = new SetPublisherAssertionsFunction();
 functions.put(SetPublisherAssertions.class.getName(),function);
```

```
    function = new ValidateValuesFunction();
    functions.put(ValidateValues.class.getName(),function);

    //  create a inspection instance
    JRInspection inspection = new JRInspection();

    // by reflexion invoke all special methods
    // these methods are added by AspectJ
    // each method adds a new function on the functions hashtable
    inspection.invokeAllSpecialMethods(this);
  }

  public final IFunction lookup(String className)
  {
      IFunction function = (IFunction)functions.get(className);
    return function;
  }
}
```

B) Code source du générateur de requêtes

Nous exposons dans cette section, un extrait detaillé et documenté du code source des servltes **JRFunctionPublisherServlet** et **JRRequestPublisherServlet**, éléments racines du processus de génération d'une nouvelle requête/réponse.

Code source du servlet **JRRequestPublisherServlet**

```
/**
 * Created on Feb 5, 2005
 * this servlets publish a new JRegistre request and response
 * A default jregistre function is published
 * @author Radhouane Ben-tamrout
 * @version 1.01
 */

package ca.uqam.jregistre.servlets;
…
public class JRRequestPublisherServlet extends HttpServlet {

public synchronized void doPost(HttpServletRequest httpRequest,
            HttpServletResponse httpResponse)throws
ServletException, IOException {

PrintWriter out = httpResponse.getWriter ();
try
{
```

```
JRListener xsdListener = new JRListener();
JRGenProcess genProcess = new JRGenProcess();
JRCompiler compiler = new JRCompiler();
JRFunctionGenerator functionGenerator = new JRFunctionGenerator();
JRJarUtils jarUtils = new JRJarUtils();
JRAspectGenerator aspectGenerator = new JRAspectGenerator();
JRAspectCompiler aspectCompiler = new JRAspectCompiler();
JRIntegrator integrator = new JRIntegrator();

// makes a copy of all the initial jars
jarUtils.copyFile(JREGISTRE_COMMONS_JAR,PROXY_COMMONS_JAR);
jarUtils.copyFile(JREGISTRE_COMMONS_JAR,DEVELOPPER_COMMONS_JAR);
jarUtils.copyFile(JREGISTRE_FUNCTION_JAR,DEVELOPPER_FUNCTION_JAR);
jarUtils.copyFile(JREGISTRE_FUNCTION_JAR,ECLIPSE_FUNCTION_JAR);
jarUtils.copyFile(JREGISTRE_JUDDI_PROXY,ECLIPSE_JUDDI_PROXY);

// creates the XSD request file and puts the content
// of httpRequest.getParameter("request")in the request file

xsdListener.createFile(XSD_REQUEST,httpRequest.getParameter("request
"));

//creates the XSD response file and puts the content
//of httpRequest.getParameter("response")in the response file

xsdListener.createFile(XSD_RESPONSE,httpRequest.getParameter("respon
se"));
// genetates the JRegistre request java class (or classes) and this
handler (or these  handlers).
JRUserClass request =(JRUserClass)genProcess.execute(XSD_REQUEST,
REQ_PACK,SOURCES_DIRECTORY);
//genetates the JRegistre response java class (or classes) and this
handler (or these handlers).
JRUserClass response =(JRUserClass)genProcess.execute(XSD_RESPONSE,
RES_PACK,SOURCES_DIRECTORY);

//generates the JRegistre function skeleton to help the request
publisher to develop the function
JRFunction function = functionGenerator.execute(request,response,
FUN_PACK,SOURCES_DIRECTORY);

//  generates the request aspect which must be able to:
//  add the request and the response to the <<IJRegistre>> interface
//  add the request and the response to the <<JRAbstractJRegistre>>
//  add the request and response handlers to <<JRHandlerMaker>>

String requestAspectPath =
aspectGenerator.generateRequestAspect(function,ASPECT_DIRECTORY) ;

//  generates the function aspect which must be able to:
//  add the function to the <<JRFunctionMaker>>
```

```
String functiontAspectPath =
aspectGenerator.generateFunctionAspect(function,ASPECT_DIRECTORY) ;

// Compiles all generated java source

String [] requestClasses   =
compiler.compile(request.getSourceCode(),SOURCES_DIRECTORY,CLASSES_D
IRECTORY,CLASSPATH);
String [] requestHandlers  =
compiler.compile(request.getHandlerSourceCode(),SOURCES_DIRECTORY,CL
ASSES_DIRECTORY,CLASSPATH);
String [] responseClasses  =
compiler.compile(response.getSourceCode(),SOURCES_DIRECTORY,CLASSES_
DIRECTORY,CLASSPATH);
String [] responseHandlers =
compiler.compile(response.getHandlerSourceCode(),SOURCES_DIRECTORY,C
LASSES_DIRECTORY,CLASSPATH);
String [] functionClasses  =
compiler.compile(function.getSourceCode(),SOURCES_DIRECTORY,CLASSES_
DIRECTORY,CLASSPATH);

//adds the new request and response and these handlers
//to the jar which contains all the old request and response

String[] commonsEntries =
JRServletUtils.mergeEntries(requestClasses,responseClasses,requestHa
ndlers,responseHandlers);
jarUtils.updateJar(JREGISTRE_COMMONS_JAR,TEMP_COMMONS_JAR,commonsEnt
ries,CLASSES_DIRECTORY);

//adds the function skeleton
//to the jar which contains all the old functions

jarUtils.updateJar(JREGISTRE_FUNCTION_JAR,TEMP_FUNCTION_JAR,function
Classes,CLASSES_DIRECTORY);

// Apply the request aspect to the jar which contains
// the new request and response and these handlers
// this aspect will modify these classes:
//add the request and the response to the <<IJRegistre>> interface
//add the request and the response to the <<JRAbstractJRegistre>>
//add the request and response handlers to <<JRHandlerMaker>>
// add a new classes to be able to execute the new modification

aspectCompiler.compileAspect(requestAspectPath,TEMP_COMMONS_JAR,ASPE
CT_COMMONS_JAR,ASPECT_CLASSPATH);

//Apply the function aspect to the jar which contains
// the new function
//this aspect will add the function to the <<JRFunctionMaker>>
```

```
aspectCompiler.compileAspect(functiontAspectPath,TEMP_FUNCTION_JAR,A
SPECT_FUNCTION_JAR,
ASPECT_CLASSPATH+":"+ASPECT_COMMONS_JAR);

// copy the new jars generated by the request Aspect and function
Aspect on the JRegistre libraries (lib directory). Tomcat it able to
reload the changed libraries. We dont have to create a new
classLoader to load the new added classes

jarUtils.copyFile(ASPECT_COMMONS_JAR,JREGISTRE_COMMONS_JAR);
jarUtils.copyFile(ASPECT_FUNCTION_JAR,JREGISTRE_FUNCTION_JAR);

// copy the new jar which contains the new added request on the
proxy directory. All JRegistre clients download the proxy from this
directory

jarUtils.copyFile(ASPECT_COMMONS_JAR,PROXY_COMMONS_JAR);

//copy the new jar which contains the new added request on the
developper directory
//the function developper must know the parameter and the return
value of the function to develop

jarUtils.copyFile(ASPECT_COMMONS_JAR,DEVELOPPER_COMMONS_JAR);
jarUtils.copyFile(ASPECT_COMMONS_JAR,ECLIPSE_COMMONS_JAR);
...
}
```

Code source du servlet **JRFunctionPublisherServlet**

```
/**
 * Created on Feb 25, 2005
 * this servlets publish a new function for the new request
 * @author Radhouane Ben-tamrout
 * @version 1.01
 */
package ca.uqam.jregistre.servlets;

import java.io.File;
import java.io.PrintWriter;
import java.io.IOException;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.ServletException;
import java.util.Hashtable;
import ca.uqam.jregistre.listener.JRListener;
import ca.uqam.jregistre.genProcess.JRFunction;
import ca.uqam.jregistre.compiler.JRCompiler;
import ca.uqam.jregistre.integrator.JRIntegrator;
import ca.uqam.jregistre.util.JRServletUtils;
```

```
import ca.uqam.jregistre.util.JRProperties;
import ca.uqam.jregistre.util.JRJarUtils;

public class JRFunctionPublisherServlet extends HttpServlet {

      String SERVER_HOME = System.getProperty("catalina.home");

public void doPost(HttpServletRequest httpRequest,
            HttpServletResponse httpResponse)throws
                        ServletException, IOException {

PrintWriter out = httpResponse.getWriter ();
final String COMMONS_JAR = "jregistre-commons-part.jar"    ;
final String FUNCTION_JAR = "jregistre-function-part.jar";
final String REQUEST = "request.xsd";
final String RESPONSE= "response.xsd";
final String FUNCTIONS = "temp-functions.bin";

JRProperties.load(SERVER_HOME+"/webapps/jregistre/WEB-
INF/","jregistre.properties");
final String
SOURCES_DIRECTORY=JRProperties.getStringProperty("jregistre.sources"
);
final String
CLASSES_DIRECTORY=JRProperties.getStringProperty("jregistre.classes"
);
final String JREGISTRE_FUNCTION_JAR =
JRProperties.getStringProperty("jregistre.lib.function");
final String
DEVELOPPER_DIRECTORY=JRProperties.getStringProperty("jregistre.devel
opper");
final String
DEVELOPPER_FUNCTION_JAR=DEVELOPPER_DIRECTORY+FUNCTION_JAR;

final String TEMP_DIRECTORY =
JRProperties.getStringProperty("jregistre.temp");
final String TEMP_FUNCTION_JAR = TEMP_DIRECTORY+FUNCTION_JAR;
final String TEMP_FUNCTIONS = TEMP_DIRECTORY+FUNCTIONS;

String CLASSPATH =
JRProperties.getStringProperty("jregistre.classpath");
JRListener functionListener = new JRListener();
JRCompiler compiler = new JRCompiler();
JRIntegrator integrator = new JRIntegrator();
JRJarUtils jarUtils = new JRJarUtils();
Hashtable tempFunctions= new Hashtable();
try{
String functionName = httpRequest.getParameter("functionName");
File tempFunctionsFile = new File(TEMP_FUNCTIONS);
tempFunctions = JRServletUtils.load(tempFunctionsFile);
JRFunction function =(JRFunction)tempFunctions.get(functionName);
```

```
String functionContent = httpRequest.getParameter("function");

//creates the java source file and puts the content
//of httpRequest.getParameter("function")in this file.

functionListener.createFile(function.getFunctionFile().getAbsolutePa
th(),functionContent);

// Compiles the function java source

String [] functionClasses  =
compiler.compile(function.getSourceCode(),SOURCES_DIRECTORY,CLASSES_
DIRECTORY,CLASSPATH);

// Update the jar which by adding into the new function

jarUtils.updateJar(JREGISTRE_FUNCTION_JAR,TEMP_FUNCTION_JAR,function
Classes,CLASSES_DIRECTORY);

// copy the new jar that contains the new function on the JRegistre
libraries (lib directory). Tomcat it able to reload the changed
libraries. We dont have to create a new classLoader to load the new
added classes

jarUtils.copyFile(TEMP_FUNCTION_JAR,JREGISTRE_FUNCTION_JAR);
…
…
      }
catch(Exception e)
      {     out.println("<html>");
            out.println("<head>");
            out.println("<title> la fonction </title>");
            out.println("</head>");
            out.println("<body>");
            out.println("</body>");
            out.println("</html>");
            }
}
}
```

N.B: Pour plus de détails sur le code source consulter le CD-ROM ci-joint.

ANNEXE B :
DEPLOIEMENT DE JREGISTRE

A) Déploiment de JRegistre sur le serveur Tomcat

Dans cette section nous expliquons en détail les étapes nécessaire pour l'installation de JRegistre. Cette installation est spécifique au système UNIX. Nous supposons qu'un serveur Tomcat est déja installé. Pour plus de détails sur les étapes d'installation de ce serveur consulter le site web officiel du serveur Tomcat (*http://tomcat.apache.org*). Nous supposons aussi qu'un registre jUDDI est dèja deployé localement sur le même serveur ou sur un serveur distant. Pour plus de détails sur les étapes d'installation du registre jUDDI consulter le site officiel du registre jUDDI (*http://ws.apache.org/juddi/*).

Le déploiement de JRegistre sur un serveur Tomcat passe par les étapes suivantes :

- *Étape 1* : téléchargez le fichier *JRegistre.zip* contenant JRegistre. Ensuite désarchiver dans le répertoire *webapps* de Tomcat.

- *Étape 2* : ajouter le bout de code suivant dans le fichier *server.xml* présent dans le répértoire *conf* de Tomcat.

```
<Context path="/jregistre" docBase="jregistre"
            debug="5" reloadable="true"
            crossContext="true">
</Context>
```

- *Étape 3* : Modifier le fichier *jregistre/WEB-INF/jregistre.properties*. La propriété `jregistre.home` doit contenir le nouveau chemin de JRegistre sur le serveur Tomcat.

- *Étpae 4* : Modifier le fichier *jregistre/WEB-INF/juddi.properties*. Ce fichier contient le chemin du registre jUDDI avec lequel JRegistre doit communiquer. Voici les propiétés qui doivent être mis à jour :

a. `juddi.proxy.adminURL=`*`adresse du registre jUDDI`*`/admin`

b. `juddi.proxy.inquiryURL=`*`adresse regis jUDDI`*`/inquiry`

c. `juddi.proxy.publishURL=`*`adresse Regis jUDDI`*`/publish`

- *Étape 5* : Démarrer JRegistre. Tomcat offre une console pour démarrer les application qui sont installer sur ce dernier. Une fois le registre est démarré, toutes les librairies (archives) utilisées par le registre seront chargées automatiquement. Le répertoire *\jregistre\WEB-INF\lib* contient l'ensmble de ces librairies y compris les librairies propre à JRegistre.

Jusque là, le registre est totalement deployé sur le serveur Tomcat. Il devient disponible pour utilisation.

B) Utilisation de JRegistre par les clients

Les clients publicateurs de requêtes et les autres clients qui veulent utiliser le registre doivent avoir une copie des librairies **`jregistreProxy.jar`**, **`jregistre-commons-part.jar`** et **`juddi-proxy.jar`**. Ces libraires sont récuperable sur le site web du registre. Les clients doivent aussi créer un fichier *properties* sous le nom de proxy.properties qui contient la ligne suivante :

`jregistre.proxy.url = `*`URL de JRegistre`*`/receiver`

Cette ligne indique le point d'entrée à JRegistre. Elle permet au proxy de localiser le registre JRegistre avec lequel on veut communiquer.

BIBLIOGRAPHIE

Akkiraju R., Flaxer D., Chang H., Chao T., Zhang L.J., Wu F. et Jeng J.J., « A Framework for Facilitating Dynamic e-Business Via Web Services », *Proceedings of the OOPSLA Workshop*, Tampa, 2001.

Akkiraju R., Goodwin R. et Roeder P.D.S., « A Method for Semantically Enhancing the Service Discovery Capabilities of UDDI », *Proceedings of IJCAI Information Integration on the Web Workshop*, p. 87-92, Acapulco, Mexico, Août 2003.

BPEL4WS: « Business Process Execution Language for Web Services », Version1.0 <ftp://www6.software.ibm.com/software/developer/library/ws-bpel1.pdf>, juillet 2002.

Chen Z., Chia L.T., Silverajan B. et Lee B.S., « UX- An Architecture Providing QoS-Aware and Federated Support for UDDI », *Proceedings of the International Conference on Web Services, ICWS '03*, p. 171-176, Las Vegas, 23-26 juin 2003.

Colgrave J., Akkiraju R. et Goodwin R., « External Matching in UDDI », *proceedings of IEEE International Conference on Web Services (ICWS)*, p.226, San Diego, juillet 2004.

EMF-XSD: «XML Schema Infoset Model (XSD)» proposed by Eclipse Modeling Framework (EMF), < http://eclipse.org/emf/xsd.php >,2002.

jUDDI: «Java implementation of the Universal Description, Discovery, and Integration (UDDI)», Apache,< http://ws.apache.org/juddi/ >, 2003.

Kiczales G., Hilsdale E., Hugunin J., Kersten M., Palm J. et Griswold W.G., « An Overview of AspectJ », *Proceedings of the 15th European Conference on Object-Oriented Programming ECOOP'2001,* p. 327-353, Budapest, Hungary, 18-22 juin 2001.

Kiczales G., Lamping J., Mendhekar A., Maeda, C., Videira Lopes C., Loingtier J.M. et Irwin J., « Aspect-Oriented Programming », *Proceedings of the European Conference on Object-Oriented Programming ECOOP'1997*, p.220, Jyväskylä, Finnland, juin 1997.

Ludwig H., « Web Services QoS: External SLAs and Internal Policies: Or, How Do We Deliver What We Promise? », *4th IEEE International Conference on Web Information Systems Engineering (WISEW'03), IEEE CS Press*, p. 115-120, 2003.

Martin D., M. Paolucci M., McIlraith S., Burstein M., McDermott D., McGuinness D., Parsia B., Payne T., Sabou M., Solanki M., Srinivasan N. et Sycara K., « Bringing Semantics to Web Services: The OWL-S Approach » *First International Workshop on Semantic Web Services and Web Process Composition SWSWPC'2004*, p. 90-93, San Diego, 2004.

Mili H., ben tamrout R. et Obaid A., « JRegistry: an Extensible UDDI Registry », *publié dans les actes de NOuvelles TEchnologies de la Répartition (NOTERE 2005),* p. 115-128, Ottawa 29 août -1er sept. 2005.

OASIS-UDDI : «Universal Description, Discovery, and Integration (UDDI) », UDDI Version 2.04 API Specification and UDDI, Version 2.03, Data Structure Reference, <http://www.oasis-open.org/committees/uddi-spec/doc/tcspecs.htm#uddiv2>, 19 juillet 2002.

OWL-S: «Semantic Markup for Web Services », a white paper describing the key elements of OWL-S, < http://www.daml.org/services/owl-s/1.0/owl-s.pdf >, 2003.

OWL-S/UDDI: « Semantic Web Services tools, The OWL-S/UDDI registry and OWL-S matchmaking functionalities », <http://www.daml.ri.cmu.edu/tools/index.html>, 2004.

Paolucci M., Srinivasan N., Sycara K. et Nishimura T., « Towards a Semantic Choreography of Web Services: From WSDL to DAML-S », *Proceedings of first International Conference on Web Services, ICWS '03*, p. 22-26, Las Vegas, 23-26 juin 2003.

Pilioura T., Tsalgatidou A. et Batsakis A., « Using WSDL/UDDI and DAML-S in web service discovery », *Proceedings of the WWW 2003 Workshop on E-Services and the Semantic Web*, Budapest , 20 mai 2003.

Ran S., « A Model for Web Services Discovery With QoS », *ACM SIGecom Exchanges*, Vol. 4, No. 1, p. 110, mars 2003.

ShaikhAli A., Rana O., Al-Ali R. et Walker D.W., « UDDIe: An Extended Registry for Web Services », *Proceedings of the Service Oriented Computing: Models, Architectures and Applications, SAINT'2003 IEEE Computer Society Press*, p. 85-90, Oralndo, janvier 2003.

Sirin E., Hendler J. et Parsia B., « Semi-automatic Composition of Web Services using Semantic Descriptions », *Proceedings of the Workshop on Web Services: Modeling, Architecture and Infrastructure in conjunction with ICEIS2003*, p. 286-291, 2003.

Sivashanmugam K., Verma K., Sheth A. et Miller J., « Adding Semantics to Web Services Standards », *Proceedings of the 1st International Conference on Web Services (ICWS'03)*, p. 395-401. Las Vegas, juin 2003.

Srinivasan N., Paolucci M. et Sycara K., « Adding OWL-S to UDDI: implementation and throughput », *Proceedings of 1st Intl. Workshop on Semantic Web Services and Web Process Composition SWSWPC'2004*, p. 6-9, 2004.

Srivastava B. et Koehler J., « Web Service Composition - Current Solutions and Open Problems », *ICAPS 2003 Workshop on Planning for Web Services*, p 28-35, 2003.

Tian M., Gramm N., Naumowicz T., Ritter H. et Schiller J., « A Concept for QoS Integration in Web Services », *1st Web Services Quality Workshop (WQW 2003), in conjunction with 4th International Conference on Web Information Systems Engineering (WISE 2003)*, p. 149 -155, Rome, Italie, décembre 2003.

W3C-nameSpaces: « Namespaces in XML », Version 1.1, W3C Recommendation, <http://www.w3.org/TR/xml-names11/>, 4 février 2004.

W3C-SOAP: « Simple Object Access Protocol (SOAP) », Version 1.1, W3C Note, <http://www.w3.org/TR/2000/NOTE-SOAP-20000508/>, 8 mai 2000.

W3C-Web Services Architecture: « Web Services Architecture », W3C Working Group Note, <http://www.w3.org/TR/ws-arch/>, 11 Février 2004.

W3C-WSDL: « Web Services Description Language (WSDL) », Version 1.1, W3C Note, <http://www.w3.org/TR/wsdl>, 15 mars 2001.

W3C-XML: « Extensible Markup Language (XML) », version 1.0, W3C Recommendation, < http://www.w3.org/TR/2004/REC-xml-20040204>, 4 Février 2004.

W3C-XSD: « XML Schema Part 0: Primer », Version 1.0, W3C Working Draft, <http://www.w3.org/TR/2000/WD-xmlschema-0-20000225>, 25 février 2000.

Zhang L.J., Bing L., Chao T. et Chang H.Y., « On Demand Web Services-Based Business Process Composition », *IEEE International Conference on Systems, Man & Cybernetics SMC'03,* octobre 2003.

Zhang L.J., H., Chang H. et Chao T., « XML-based advanced UDDI search mechanism for B2B integration », *Proceedings of the Fourth IEEE International Workshop on Advanced Issues of E-Commerce and Web-Based Information Systems, IEEE CS Press,* p.9-16, 2002.

Zhang L.J., Chang H. et Chao T., « Web Services Relationships Binding for Dynamic e-Business Integration », *International Conference on Internet Computing (IC'02),* p. 561-570, Las Vegas, 24-27 mai 2002.

Zhang L.J., Mao Z.H. et Li Y.D., « Mathematical Analysis of Mutation Operator in Genetic Algorithms and Its Improved Strategy », *International Conference on Neural Computing,* Beijing, 1995.